Université du Québec à Chicoutimi

Mémoire présenté à
l'Université du Québec à Chicoutimi
comme exigence partielle
de la maîtrise en informatique

offerte à

l'Université du Québec à Chicoutimi
en vertu d'un protocole d'entente
avec l'Université du Québec à Montréal

par

MOHAMED ANOUAR TALEB

Parallélisation d'un algorithme génétique pour le problème d'ordonnancement sur machine unique avec temps de réglages dépendants de la séquence

Avril 2008

UNIVERSITÉ DU QUÉBEC À MONTRÉAL
Service des bibliothèques

Avertissement

La diffusion de ce mémoire se fait dans le respect des droits de son auteur, qui a signé le formulaire *Autorisation de reproduire et de diffuser un travail de recherche de cycles supérieurs* (SDU-522 – Rév.01-2006). Cette autorisation stipule que «conformément à l'article 11 du Règlement no 8 des études de cycles supérieurs, [l'auteur] concède à l'Université du Québec à Montréal une licence non exclusive d'utilisation et de publication de la totalité ou d'une partie importante de [son] travail de recherche pour des fins pédagogiques et non commerciales. Plus précisément, [l'auteur] autorise l'Université du Québec à Montréal à reproduire, diffuser, prêter, distribuer ou vendre des copies de [son] travail de recherche à des fins non commerciales sur quelque support que ce soit, y compris l'Internet. Cette licence et cette autorisation n'entraînent pas une renonciation de [la] part [de l'auteur] à [ses] droits moraux ni à [ses] droits de propriété intellectuelle. Sauf entente contraire, [l'auteur] conserve la liberté de diffuser et de commercialiser ou non ce travail dont [il] possède un exemplaire.»

Résumé

Les problèmes d'ordonnancement peuvent être rencontrés dans plusieurs situations de la vie courante. Organiser des activités quotidiennes, planifier un itinéraire de voyage sont autant d'exemples de petits problèmes d'optimisation que nous tentons de résoudre tous les jours sans nous en rendre compte. Mais quand ces problèmes prennent des proportions plus grandes, il devient difficile au cerveau humain de gérer tous ces paramètres et le recours à une solution informatique s'impose. Les problèmes d'ordonnancement en contexte industriel sont nombreux et celui qui retient particulièrement notre attention dans le cadre de ce mémoire est le problème d'ordonnancement de commandes sur machine unique avec temps de réglages dépendant de la séquence. Ce problème fait partie de la classe de problèmes NP-Difficiles.

Etant donnée sa complexité, ce problème ne peut être résolu par une méthode exacte. Les métaheuristiques représentent ainsi une bonne alternative pour obtenir des solutions de bonne qualité dans des délais très courts. Les algorithmes génétiques, qui font partie des algorithmes évolutionnaires, sont utilisés dans ce travail pour résoudre ce problème d'ordonnancement.

La prolifération des architectures parallèles a ouvert la voie à un nouvel éventail d'approches pour optimiser les algorithmes et plus spécialement les métaheuristiques. Ce mémoire propose une stratégie de parallélisation de l'algorithme génétique pour en étudier les bénéfices.

Le premier algorithme génétique proposé est implémenté sur le modèle d'un algorithme de la littérature. Cet algorithme ne s'est pas avéré performant pour toute la série de problèmes test et, pour cette raison, des modifications de paramètres ont été rendues nécessaires. Ces modifications ont donné naissance à une deuxième version séquentielle dont les résultats se sont avérés satisfaisants. Une troisième version a ensuite été implémentée avec une optique d'exécution parallèle selon un modèle en îlot et une topologie en anneau unidirectionnel. Un plan d'expérience a ensuite été mis au point selon plusieurs variables et vise à identifier les meilleures configurations de l'algorithme tant sur le plan de la qualité des résultats que sur le plan de l'accélération.

Les résultats obtenus dans ce mémoire montrent que l'introduction de la parallélisation dans un algorithme génétique est bénéfique à ce dernier tant sur le plan qualité des résultats que sur le plan accélération. Dans un premier temps, la version sans communications n'a pas

amélioré une grande partie des problèmes mais a pu atteindre des accélérations linéaires. Par la suite, l'introduction des échanges a nettement influé sur la qualité des résultats. En effet, en adoptant une stratégie de division de la taille de la population par le nombre de processeurs, l'algorithme génétique parallèle parvient à donner des résultats équivalents voire meilleurs que la version séquentielle, et ceci pour plusieurs fréquences d'échanges entre les populations.

Remerciements

J'adresse tout d'abord mes plus grands remerciements à mon directeur de recherche, monsieur Marc Gravel pour son support, ses conseils et sa patience qui m'ont été très précieux tout au long de cette aventure.

Je remercie également mon professeur et ami Pierre Delisle. C'est un privilège pour moi d'avoir côtoyé une personne aussi talentueuse et passionnée.

Je voudrais aussi remercier les membres du groupe de recherche en informatique (GRI) à commencer par sa coordonnatrice madame Caroline Gagné ; sans oublier les étudiants : Sébastien Noêl, Jêrome Lambert, Etienne Lafrenière, Arnaud Zinflou, Jean-Michel Gilbert, Patrice Roy et Kaven Breton.

Mes remerciements vont aussi aux professeurs du département d'informatique et de mathématiques de l'université du Québec à Chicoutimi pour leur accueil chaleureux et leur dévouement aux étudiants. Je remercie également le centre universitaire de recherche sur l'aluminium (CURAL) pour avoir à disposition leurs ordinateurs parallèles.

Ce mémoire est dédié à ma famille : mon père Mohamed, ma mère Bolbol dont la fierté est ma raison de vivre ; mon frère Mohamed Ali, ma soeur Nour El Houda, mon oncle Seif et ma grand-mère Noura. Rien de tout ceci ne serait possible sans leurs sacrifices et leurs encouragements.

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Introduction

Le parallélisme est un concept de plus en plus populaire et contribue grandement à satisfaire la demande de performance croissante en informatique. Considérées à leurs débuts comme réservées à une élite, les machines parallèles sont, de nos jours, abordables et accessibles et la technologie pour les manipuler est devenue beaucoup plus conviviale et performante. Cette prolifération rapide de la technologie parallèle a atteint beaucoup de domaines parmi lesquels l'optimisation combinatoire.

Des exemples de problèmes d'optimisation combinatoire peuvent être retrouvés partout dans des secteurs industriels ou de services. La résolution de ces problèmes peut souvent se faire à l'aide de méthodes exactes. Cependant, pour une classe de problèmes dits NP-Difficiles, il n'est pas possible de trouver la solution optimale. Les métaheuristiques représentent alors une alternative pour résoudre ces problèmes et sont reconnues pour donner des résultats très satisfaisants dans un temps acceptable.

Entre autres, la simplicité d'implémentation des métaheuristiques ainsi que leur grande capacité d'adaptation aux divers problèmes en ont fait une alternative de choix auprès des chercheurs pour la résolution des problèmes d'ordonnancement. Ce succès se traduit par les nombreuses recherches, travaux, publications et conférences exclusivement réservés à l'amélioration et à l'innovation dans ce domaine.

Ce mémoire s'intéresse particulièrement à l'utilisation des algorithmes génétiques (AG) dans leur version parallèle pour la résolution du problème d'ordonnancement de commandes sur machine unique avec temps de réglages dépendant de la séquence.

Le plan de ce mémoire est le suivant : Le Chapitre 2 introduit les domaines du parallélisme et des métaheuristiques. Dans un premier temps, le domaine du parallélisme est introduit sur le plan de l'architecture et de la conception d'algorithmes. Une revue de littérature est faite sur les avancées matérielles et logicielles suivie d'une présentation des mesures de performance. Ensuite, les métaheuristiques séquentielles et parallèles sont présentées avec une revue des principales métaheuristiques de la littérature, leurs applications et les principaux travaux qui ont été faits pour chacune d'elles. Les objectifs de la recherche sont présentés à la fin de ce chapitre. Le premier objectif consiste à mettre au point une version séquentielle de l'algorithme génétique et de comparer ses résultats avec ceux d'un autre algorithme de la littérature. Le deuxième objectif consiste à transformer cet algorithme en une version parallèle et d'étudier les résultats obtenus en variant ses paramètres.

Le Chapitre 3 présente de façon détaillée la phase de conception. D'abord, le problème d'ordonnancement sur machine unique avec temps de réglages dépendants de la séquence est énoncé. Les sous-sections de ce chapitre décrivent les différentes phases du plan d'expérience défini. Ce plan consiste à la conception et l'implémentation d'un algorithme génétique séquentiel basé sur celui de Rubin & Ragatz [134] et est suivie de la conversion de cet algorithme séquentiel en un algorithme parallèle. Par la suite, une série d'essais numériques est présentée pour l'identification des meilleurs paramètres et configurations. Il a été constaté d'après ces essais que le comportement de l'algorithme diffère selon la taille du problème. De plus, l'approche adoptée de subdivision de l'espace de recherche est un facteur déterminant sur la qualité des solutions et sur le temps d'exécution.

Le Chapitre 4 vient conclure la recherche en rappelant les objectifs définis et en démontrant de quelle façon ils ont été atteints.

Chapitre 2

Le parallélisme et les métaheuristiques

2.1 Le parallélisme

2.1.1 Introduction

Dans un domaine comme l'informatique, les besoins de performance sont toujours croissants. Les programmes informatiques doivent gérer de plus en plus de données et ce, de plus en plus rapidement. Il est toutefois possible de rendre un programme plus performant en améliorant l'algorithme ou la machine sur laquelle il s'exécute. Une autre approche consiste à combiner les deux options et ainsi adapter un programme à une architecture matérielle plus performante. C'est pour répondre à une demande de plus en plus pressante qu'est apparu le parallélisme.

Traditionnellement, la puissance des ordinateurs a été augmentée en apportant des améliorations dans le matériel et plus particulièrement les circuits électroniques. Grâce notamment à la technologie VLSI (Very Large Scale Integration) qui a permis des avancées spectaculaires dans le domaine des microprocesseurs. À titre d'exemple, les fréquences d'horloge sont passées de 40Mhz (fin des années 80), 2Ghz (2002) [73] jusqu'à 3.6Ghz (2006) [21]. Même si les limites sont loin d'être atteintes, il est constaté que chaque nouvelle amélioration nécessite de plus en plus d'efforts scientifiques et d'investissements financiers. Les concepteurs de processeurs sont ainsi constamment à la recherche d'alternatives pour donner aux processeurs encore plus de puissance. Le parallélisme offre une alternative à cette escalade en mettant côte à côte les processeurs d'une même génération et en tirer un meilleur profit. Ceci se traduit par

un accès multiple aux unités de stockage, des performances extensibles (scalable performance) et des coûts réduits.

D'un point de vue historique, le calcul haute performance en général et le calcul parallèle en particulier, relevaient du domaine spécialisé de l'industrie informatique servant les besoins de la science, de l'ingénierie et de la défense. Des pionniers du parallélisme comme Dennis *et al.* [43] ou Schwartz [136] ont exposé les principes fondamentaux au cours des années soixante-dix. Dennis *et al.* ont publié la première description d'un processeur qui exécute des programmes parallèles présentés sous forme de flux de données tandis que Scwartz a décrit et analysé un nouveau modèle d'ordinateur baptisé "Ultracomputer" dans lequel les processeurs sont reliés par un réseau d'interconnexion. Cependant, ce n'est qu'au milieu des années quatre-vingt que le domaine a connu une importante expansion avec la mise en commercialisation des premières machines parallèles à usage général [121]. Suite à cela, la programmation parallèle est apparue par opposition à la programmation séquentielle "traditionnelle", permettant de contrôler quelles instructions peuvent être exécutées en parallèle sur quels processeurs. Le calcul parallèle, en réduisant le temps d'exécution des programmes, a permis de mettre à portée des calculs qui étaient auparavant impossibles à faire dans des domaines comme la météorologie, la simulation nucléaire, la physique quantique et le génie génétique.

Selon Quinn [125], le calcul parallèle est "le recours à un ordinateur parallèle dans le but de réduire le temps nécessaire à la résolution d'un problème séquentiel". Ainsi, le concept de parallélisme est étroitement lié au matériel utilisé. Selon le même auteur, un ordinateur parallèle est "un ordinateur disposant de plusieurs processeurs et supportant la programmation parallèle". Il fait par conséquent un lien supplémentaire au niveau de la programmation parallèle qu'il définit comme étant "programmer en un langage qui permet une indication explicite sur quelles portions du programme seront exécutées de manière concurrente par les différents processeurs". Le parallélisme possède donc des implications matérielles, algorithmiques et de programmation. Ces différents aspects font l'objet de la section qui suit. En premier lieu, l'environnement matériel dans lequel le domaine du parallélisme évolue est présenté et est suivi par un aperçu de la conception de programmes parallèles selon un modèle particulier. Par la suite, on traite

des standards de la programmation parallèle et des mesures de performance utilisées pour évaluer les résultats. Enfin, des facteurs de performance d'algorithmes parallèles sont discutés.

2.1.2 Les différentes architectures parallèles

L'essence même du parallélisme repose sur le fait de disposer de plusieurs processeurs. Cependant, le fait d'augmenter le nombre de processeurs, de modifier leurs dispositions et de modifier l'accès à la mémoire, fait émerger autant d'architectures nouvelles. Durant plus de trois décennies, du début des années soixante jusqu'au milieu des années quatre-vingt-dix, une grande variété d'architectures parallèles ont été explorées [125]. Un des moyens pour classifier ces architectures est d'avoir recours à la taxonomie de Flynn [53] illustrée à la Figure 2.1. Cette taxonomie, basée sur l'identification et la séparation des flux d'instructions et des données, se compose de quatre principales classifications : SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data), MIMD (Multiple Instruction Multiple Data). Certains auteurs tels Duncan [49] ont proposé leur propre taxonomie mais elle n'a pas eu le même succès que celle de Flynn.

D'abord, la catégorie SISD regroupe les machines à architecture classique telle que la machine de Von Neumann [151] ainsi que la plupart des machines uni-processeurs. Dans cette catégorie, un seul processeur exécute un flux d'instructions sur un flux de données.

En deuxième lieu, la catégorie SIMD regroupe les machines disposant de plusieurs unités de calcul s'occupant chacune de leur propre flux de données. La plupart des ordinateurs SIMD opèrent de manière synchrone grâce à une horloge globale unique. Ce genre d'architecture traite les problèmes où les mêmes instructions sont appliquées sur des données régulières de grande taille.

La troisième catégorie MISD est décrite comme étant une classe de machines ne possédant pas d'exemples concrets. Elle peut être perçue en tant que plusieurs ordinateurs opérant sur un seul flux de données. Les processeurs vectoriels en "pipeline", généralement classés en tant que SISD, sont aussi considérés comme étant de type MISD [12].

Enfin, la dernière catégorie MIMD inclut les machines possédant plusieurs processeurs

opérant sur des flux d'instructions et de données distincts. Les processeurs dans cette catégorie peuvent communiquer à travers une mémoire globale partagée ou par passage de messages.

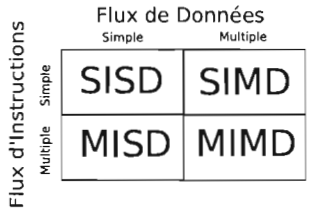

Figure 2.1 – Taxonomie de Flynn [125]

Bien que la taxonomie de Flynn soit un des meilleurs moyens connus pour catégoriser une machine parallèle [125], elle n'est pas assez précise pour décrire les machines actuelles. Alba [10] propose une extension au modèle de Flynn (Figure 2.2) où la classe MIMD est subdivisée selon l'organisation de la mémoire.

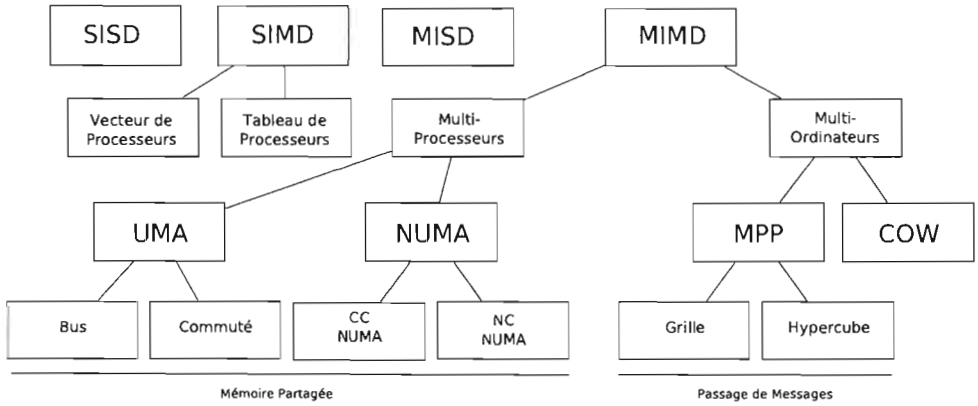

Figure 2.2 – Taxonomie de Flynn étendue [10]

D'autres taxonomies ont aussi été publiées. Les principales étant celles de Snyder [137] et Duncan [49]. La taxonomie de Flynn couvre théoriquement toutes les possibilités d'architectures parallèles. Cependant, la plupart des machines actuelles sont de type MIMD et dans cette même classe, plusieurs catégories apparaissent et font l'objet de la prochaine section.

2.1.2.1 Architectures Multi-processeurs

Lorsque, dans un ordinateur, plusieurs processeurs accèdent à une mémoire partagée, l'architecture ainsi formée est appelée multi-processeurs. On distingue deux catégories : les multi-processeurs centralisés et les multi-processeurs distribués.

Les multi-processeurs centralisés sont une extension d'un ordinateur classique puisque

tous les processeurs ont accès à une même mémoire principale et un bus relie les périphériques entrée/sortie à la mémoire partagée et relie les processeurs les uns aux autres. Chaque processeur a également accès à une mémoire cache qui réduit le temps d'attente de donnée provenant de la mémoire principale. La mémoire cache est une mémoire à accès rapide mais de petite taille, stockant les informations les plus utilisées par le processeur. Deux autres appellations sont aussi utilisées pour désigner les multi-processeurs centralisés : Uniform Memory Access (UMA) Multiprocessor et Symetric Multiprocessor (SMP). L'appellation UMA vient du fait que le temps d'accès à chaque adresse mémoire est constant parmi les processeurs et l'appellation SMP est utilisée pour indiquer l'unicité de la mémoire. Les multi-processeurs ont certains inconvénients comme le problème de cohérence de la mémoire cache. Ce problème se pose par exemple dans le cas où un processeur *P1* effectue une lecture à une adresse X de la mémoire principale à la suite d'une écriture par un processeur *P2* à la même adresse X. Si la valeur retournée par la lecture est l'ancienne valeur alors on peut dire qu'il y a problème de cohérence de la mémoire cache. Plusieurs approches ont été proposées au cours des dernières années afin de résoudre ce problème, notamment au niveau matériel [149]. Le schéma d'un multi-processeur centralisé est illustré à la Figure 2.3.

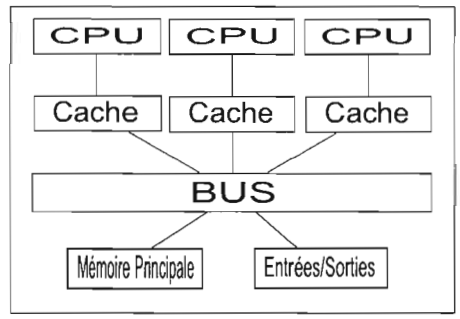

Figure 2.3 – Multi-processeurs centralisés

Le fait que, dans l'architecture précédente, la mémoire soit partagée limite le nombre de processeurs branchés en série. Ceux-ci ne dépassent pas généralement quelques douzaines [125]. Pour remédier à cela, chaque processeur est équipé de sa propre mémoire locale et les entrées/sorties sont distribuées. C'est ce qu'on appelle les multi-processeurs distribués, architecture aussi connue sous le nom de "Non Uniform Memory Access" (NUMA) (Figure

2.4). L'appellation "Non Uniform" vient du fait que le temps d'accès à la mémoire de chaque processeur n'est pas constant, par opposition à l'architecture UMA vue préalablement.

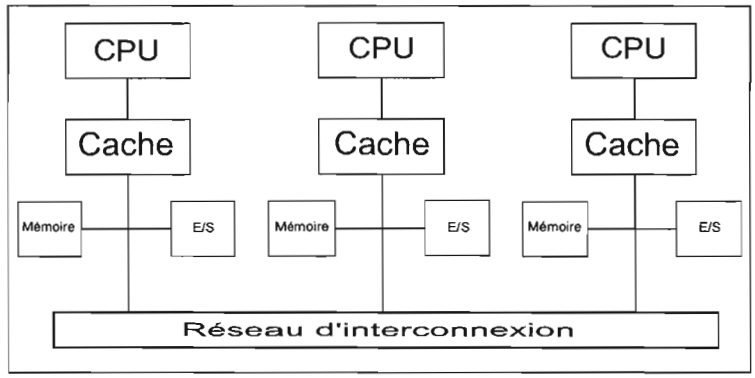

Figure 2.4 – Multi-processeurs distribués

2.1.2.2 Architectures Multi-ordinateurs

Un multi-processeurs dont les espaces d'adressage sont séparés constitue une nouvelle architecture appelée multi-ordinateurs. Puisqu'aucun espace n'est partagé par les processeurs, on évite les problèmes de cohérence de cache moyennant une gestion des communications pouvant être complexe et coûteuse. Deux grandes catégories constituent les multi-ordinateurs : multi-ordinateurs symétriques et multi-ordinateurs asymétriques.

Un multi-ordinateur asymétrique (Figure 2.5) est constitué d'un ordinateur principal et d'une série d'autres ordinateurs secondaires reliés en réseau. Ces ordinateurs utilisent des processeurs dédiés au calcul parallèle. Cette architecture a tendance à être dépendante de la machine principale et de gaspiller ses ressources lorsqu'aucune partie parallèle n'est utilisée [125]. L'appellation asymétrique est conséquente à l'unique accès de l'utilisateur via l'ordinateur principal.

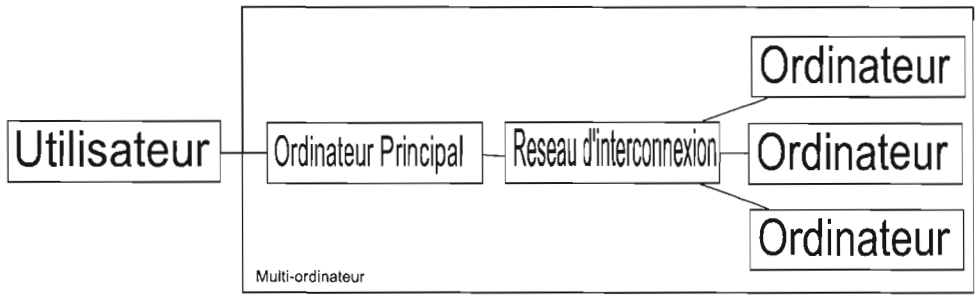

Figure 2.5 – Multi-Ordinateur asymétrique

Dans un multi-ordinateur symétrique (Figure 2.6), tous les ordinateurs exécutent la même portion du programme. Cependant, chaque noeud pouvant servir de point d'entrée/sortie, il n'est pas facile de donner l'illusion de travailler sur un seul système. Le principal souci de programmation devient la répartition équitable de la charge de travail entre tous les processeurs. L'appellation symétrique vient justement de la répartition des points d'entrée/sortie et du partage du calcul entre tous les noeuds.

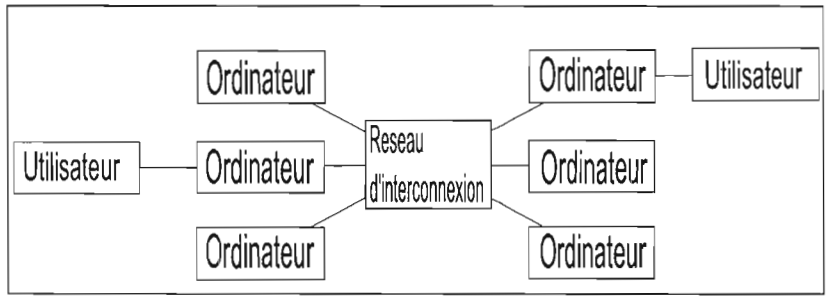

Figure 2.6 - Multi-Ordinateurs symétrique

Parmi les machines parallèles commercialisées, on cite la série Origin de SGI et, plus particulièrement, le modèle Origin 2000 [92] qui est un multiprocesseur ce type ccNUMA pouvant contenir jusqu'à 512 noeuds. Chaque noeud est composé d'un ou de deux processeurs R10000. Pour sa part, le SP2 de IBM [3] est un multi-ordinateur de type SMP qui permet de connecter un éventail de 2 à 512 noeuds. Les noeuds sont connectés par un réseau haute performance utilisant des paquets commutés et dédié à la communication inter-processeurs. Chaque noeud contient sa propre copie du système d'exploitation AIX (IBM).

Cet aperçu des architectures parallèles a permis de présenter un domaine où la diversité du matériel est importante. Chaque architecture citée précédemment possède ses avantages et ses inconvénients. Les multi-processeurs ont l'avantage de supporter de multiples utilisateurs, de ne pas perdre en efficacité lorsqu'ils traitent du code parallèle conditionnel et de disposer des mémoires caches pour réduire la charge sur le bus [125]. Par contre, les multi-processeurs sont limités par le nombre de processeurs qu'ils peuvent contenir et leur prix tend à augmenter de manière exponentielle à chaque extension [10]. Les multi-ordinateurs ont également l'avantage d'être faciles à installer et à étendre, d'être plus flexibles et d'avoir un meilleur rapport prix/performance. Cette architecture, plus précisément les multi-ordinateurs

symétriques est la plus adaptée à la programmation parallèle. Par contre, la rapidité du réseau qui relie les ordinateurs devient un facteur important de performance. Une architecture adéquate ne peut donner de bonnes performances sans des algorithmes conçus pour l'exécution parallèle. Dans la section qui suit, la notion de conception d'algorithmes parallèles va être étudiée.

2.1.3 Conception d'algorithmes parallèles

La démarche à suivre pour concevoir un algorithme parallèle diffère de celle d'un algorithme séquentiel. De nombreux travaux ont été faits dans le domaine de la conception d'algorithmes parallèles. Des auteurs comme Jàjà [87] et Grama *et al.* [73] ont énoncé des approches de conception d'algorithmes parallèles. Foster [56] propose, pour sa part, une méthodologie précise appelée "modèle tâche canal" qui est illustrée à la Figure 2.7. Ce modèle comporte quatre étapes principales : la décomposition, la communication, l'agglomération et l'assignation.

La décomposition a pour but d'identifier le plus grand nombre de sources de parallélisme possible. Cette phase fait en sorte que le problème soit décortiqué en un maximum de tâches élémentaires.

La phase de communication est abordée une fois les tâches élémentaires identifiées. Elle a pour but de les relier par des canaux de communication. C'est une étape purement liée à l'environnement parallèle parce qu'un programme séquentiel n'a pas besoin de faire communiquer ses différents composants.

La troisième étape est celle de l'agglomération. Cette phase impose des choix à faire en fonction de l'architecture parallèle adoptée. Elle consiste à rassembler les tâches primitives en tâches plus grandes, l'objectif étant de minimiser la communication entre les tâches. Un autre but de l'agglomération est d'augmenter le plus possible l'extensibilité du programme et sa portabilité.

Finalement, l'étape d'assignation consiste à attribuer les tâches aux processeurs. L'objectif de cette phase est double : répartir de façon équitable les tâches et minimiser les

communications inter-processeurs.

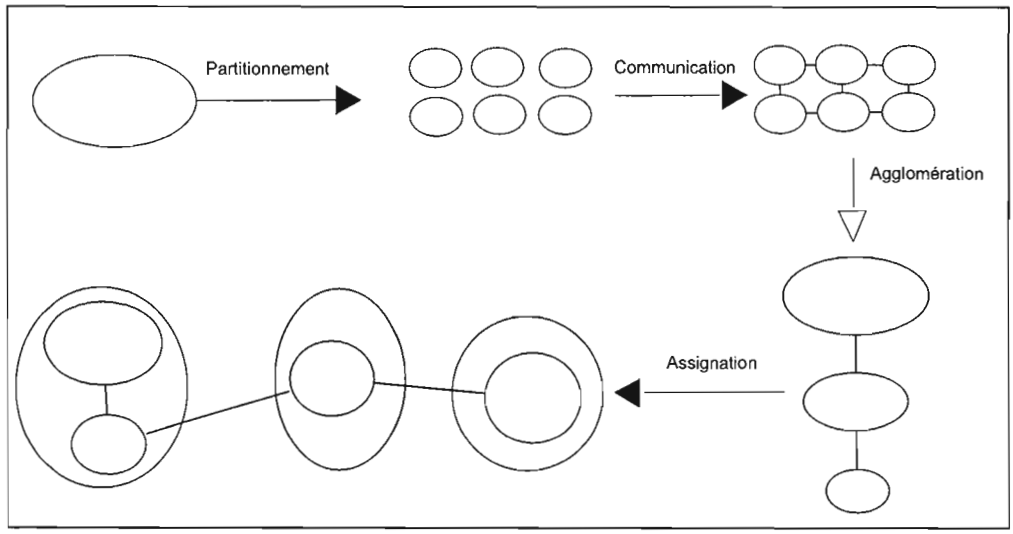

Figure 2.7 – Schéma récapitulatif du modèle tâche canal [56]

Après avoir conçu un algorithme destiné à l'exécution parallèle, l'étape suivante consiste à transposer cet algorithme en programme parallèle. Dans la prochaine section, la programmation de ces algorithmes dans des environnements parallèles sera présentée.

2.1.4 La programmation parallèle

De nombreux langages de programmation et bibliothèques ont été développés pour la programmation parallèle. Ces langages diffèrent les uns des autres par la vision de l'espace d'adressage accordée au programmeur, le degré de synchronisation et la multiplicité des programmes [73]. Les langages de programmation parallèles se basent sur des approches de conception aussi appelés "paradigmes de programmation" dont certains seront expliqués dans les sous-sections qui suivent. En premier lieu, il sera question du modèle à passage de messages et du standard MPI. Ensuite, on présentera le modèle à mémoire partagée avec le standard OpenMP. Enfin, la dernière sous-section fera un survol d'autres approches et de standards de programmation.

2.1.4.1 Modèle à passage de messages et le standard MPI

Le standard MPI (Message Passing Interface) [55] est un modèle de programmation par bibliothèques permettant l'implémentation d'un programme sur une architecture parallèle selon le modèle à passage de messages (Figure 2.8). Le modèle à passage de messages est très

similaire au modèle tâche canal discuté précédemment. Ce modèle suppose que l'environnement matériel est un ensemble de processeurs possédant chacun une mémoire locale.

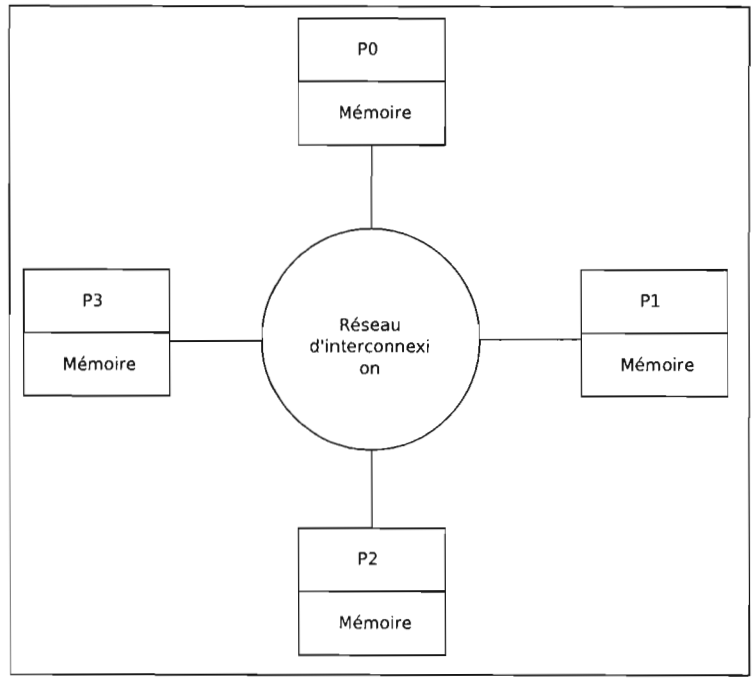

Figure 2.8 – Modèle à passage de messages [125]

Il est à préciser que le modèle à passage de messages est indépendant de l'architecture matérielle et peut être implémenté et utilisé sur une architecture à mémoire partagée. Le modèle à passage de messages repose aussi sur le principe de la parallélisation explicite [73]. En effet, le programmeur doit analyser l'algorithme ou le code séquentiel et identifier les portions qui peuvent être parallélisées. Puisque chaque processeur n'a accès qu'à sa propre mémoire, les seules interactions possibles se font par envois et réceptions de messages. Les opérations permettant l'envoi et la réception de messages sont respectivement "send" et "receive" et peuvent être soumises selon différents modes (bloquant, non bloquant, avec tampon etc.). La Figure 2.9 illustre un exemple d'envoi de messages avec ce modèle.

MPI est essentiellement apparu pour régler un problème de multitude d'interfaces à passage de messages [10]. En effet, les ordinateurs parallèles de première génération avaient des bibliothèques fournies par les constructeurs qui bien souvent n'étaient compatibles qu'avec les machines en question ou celles d'un même constructeur. Même si la différence entre les

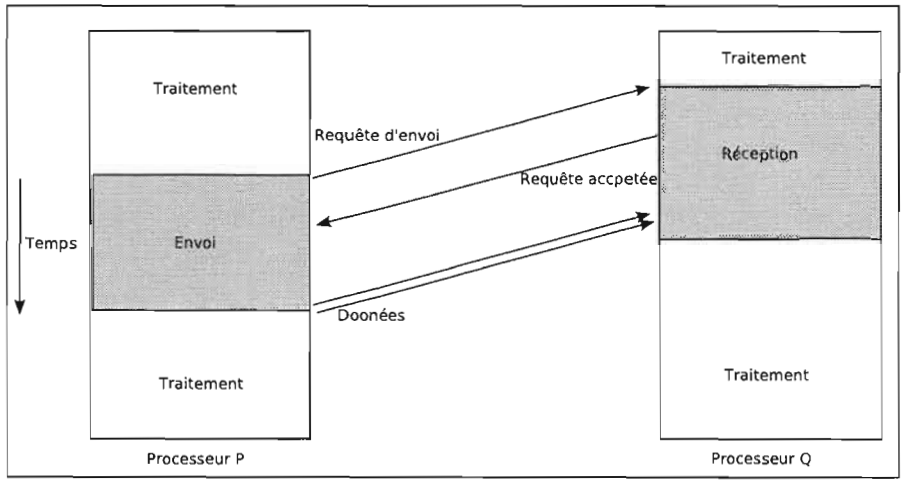

Figure 2.9 – Abstraction d'envoi de messages niveau utilisateur [150]

standards était essentiellement syntaxique, des nuances au point de vue sémantique impliquaient souvent un effort de conversion supplémentaire pour passer d'une machine à une autre. La popularité du modèle à passage de messages a créé le besoin d'unification et de portabilité et MPI a vu le jour principalement dans cette optique [23]. MPI contient plus de 125 routines des plus simples aux plus complexes.

La Figure 2.10 présente un exemple simple de programme MPI permettant d'identifier les processeurs impliqués. Dans ce programme, l'initialisation de MPI se fait par la commande MPI_Init et son arrêt par MPI_Finalize. Toutes les directives parallèles doivent être incluses entre ces deux instructions. Dans l'exemple, on fait appel à MPI_Comm_rank pour connaître le rang du processeur, MPI_Barrier pour synchroniser les processeurs et MPI_Comm_Size pour connaître le nombre total de processeurs. La deuxième partie la Figure 2.10 illustre le résultat en sortie de l'exécution sur 4 processeurs.

2.1.4.2 Modèle à mémoire partagée et le standard OpenMP

OpenMP [116] est un standard de programmation parallèle utilisant le modèle à mémoire partagée illustré à la Figure 2.11. Ce modèle est une abstraction d'un multiprocesseur centralisé. La couche matérielle est supposée être une collection de processeurs ayant chacun accès à la même mémoire. Par conséquent, les processeurs peuvent interagir par le biais de variables partagées.

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
 int id ; // rang du processeur
 int p ; // nombre de processeurs

 MPI_Init (&argc , &argv) //lancement de MPI
 MPI_Comm_rank (MPI_COMM_WORLD, &id) ; //affectation du rang du processeur dans id
 printf (" \n Processeur %d prêt pour MPI ",id) ; //affichage des rangs

 MPI_Barrier (MPI_COMM_WORLD) // barrière de synchronisation
 if ( !id) // condition pour le processeur 0
 {
   MPI_Comm_size (MPI_COMM_WORLD, &p) ; //affectation du nombre de processeurs dans p
   printf (" \n Il y a au total %d processeurs sollicités ",p) ; //affichage du nombre de processeurs
 }
 MPI_Finalize () ; // arrêt de MPI
 return 0 ;
}
```

```
Processeur 0 prêt pour MPI
Processeur 1 prêt pour MPI
Processeur 2 prêt pour MPI
Processeur 3 prêt pour MPI
Il y a au total 4 processeurs sollicités
```

Figure 2.10 – Exemple de code MPI

Il se distingue par son utilisation du modèle fork/join (Figure 2.12) où un thread principal se subdivise et se rejoint lorsque nécessaire. Concrètement, le thread maître exécute toutes les portions séquentielles du code et dès qu'il y a besoin d'une exécution en parallèle, un "fork" est enclenché ce qui engendre des thread esclaves qui démarrent à partir de ce point. Au moment où l'exécution parallèle achève, ces threads sont tués ou suspendus et le contrôle revient au thread maître, c'est ce qu'on appelle un "join".

OpenMp exécute un programme de manière séquentielle jusqu'à ce qu'il rencontre une directive de compilateur suite à laquelle il crée les threads parallèles. OpenMp est utilisé comme extension aux langages C, C++ et Fortran.

La Figure 2.13 illustre un exemple de programme OpenMP qui effectue le même traitement que l'exemple vu en MPI, à savoir identifier les processeurs présents. Dans cet exemple, le bloc d'exécution en parallèle est délimité par la directive #pragma omp parallel. L'identifiant du processeur est obtenu avec la fonction get_thread_num. La synchronisation est effectuée avec la directive #pragma_omp_barrier. Enfin, le nombre total de processeurs, qui correspond au nombre de threads, est obtenu grâce à la fonction omp_get_num_threads. Le résultat de l'exé-

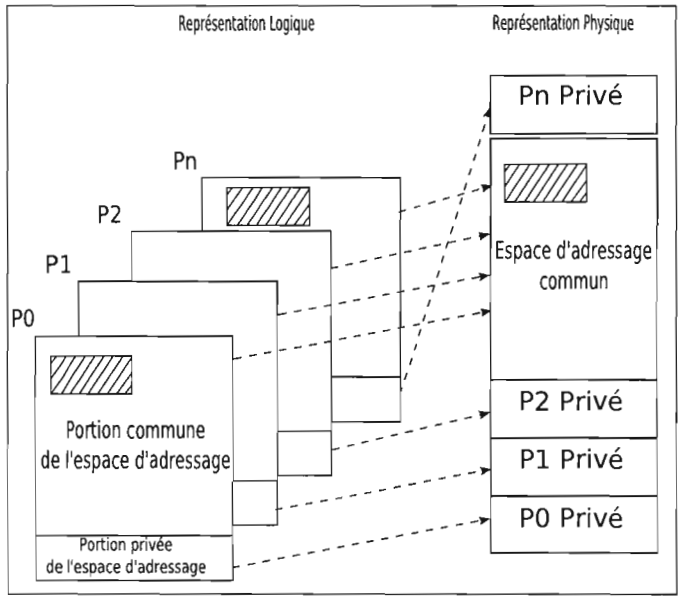

Figure 2.11 – Modèle à mémoire partagée [35]

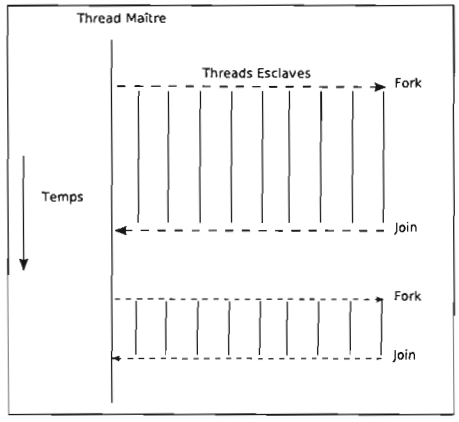

Figure 2.12 – Modèle Fork/Join [125]

cution de cet exemple sur 4 processeurs est illustré dams la deuxième partie de la Figure 2.13.

MPI et OpenMP matérialisent deux approches différentes de programmation parallèle. De par leurs abstractions matérielles différentes, chaque modèle convient à un type d'architecture. En effet, le modèle à passage de messages convient mieux aux architectures de type multi-ordinateur (processeurs ayant chacun leur mémoire et reliés par un réseau) et le modèle à mémoire partagée convient mieux aux architectures de type multiprocesseurs (processeurs accédant à la même mémoire via un bus). De plus, dans le modèle à passage de messages, les processus parallèles restent actifs tout au long de l'exécution du programme

```
#include <omp.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
 int id; // rang du thread
 int nt; // nombre de threads
 #pragma omp parallel private(id) // début du bloc parallèle
 {
   id = omp_get_thread_num(); // affectation du rang du thread dans la variable id

   printf("\n Thread %d pret pour OpenMP", id); // affichage des rangs

   #pragma omp barrier // barrière de synchronisation

   if (!id) // condition pour le thread 0
   {
     nt = omp_get_num_threads(); //affectation du nombre de threads dans nt
     printf("Il y a au total %d threads sollicités",nt); // affichage du nombre de threads
   }
 }
return 0;
}
```

```
Thread 0 prêt pour OpenMP
Thread 1 prêt pour OpenMP
Thread 2 prêt pour OpenMP
Thread 3 prêt pour OpenMP
Il y a au total 4 threads sollicités
```

Figure 2.13 – Exemple de code OpenMP

alors que dans le modèle à mémoire partagée, un thread est actif au début du programme et ce nombre peut changer dynamiquement tout au long de l'exécution. C'est un avantage considérable par rapport au modèle précédent puisqu'il permet la parallélisation incrémentale, qui est la transformation d'un programme séquentiel en un programme parallèle, un bloc de code à la fois. Un certain nombre d'autres standards de programmation parallèle existent et seront présentés dans la section suivante.

2.1.4.3 Autres standards de programmation parallèle

"Parallel Virtual Machine" (PVM) [141], comme son nom l'indique, fournit à l'utilisateur une abstraction de machine parallèle qui s'occupe de gérer les communications et autres conversions de données parmi les processeurs physiques. À l'instar de MPI, PVM est basé sur le modèle à passage de messages. L'utilisateur manipule un ensemble de tâches qui coopèrent entre elles en ayant la possibilité d'ajouter et de supprimer des ordinateurs à la machine virtuelle. La principale force de PVM est de permettre d'implémenter le modèle à passage de messages sur un réseau hétérogène. Différentes architectures et différents systèmes

d'exploitation peuvent faire partie de la machine virtuelle et tout ceci dans la plus grande transparence. Ceci est possible grâce à l'exécution d'un processus en arrière plan appelé "démon" sur chacune des machines du réseau. Le démon PVM s'occupe de l'harmonisation de la communication entre les différentes machines. De plus, PVM permet à l'utilisateur de décider explicitement quelle tâche va être attribuée à quel ordinateur de la machine parallèle.

D'autres standards de programmation parallèle sont apparus sous la forme d'extension de langages. D'ailleurs, certains auteurs comme Foster [57] considèrent MPI et OpenMP non pas comme des extensions de langages mais comme des approches à base de bibliothèques. Au lieu de définir au complet un nouveau langage, ces extensions ont l'avantage de garder la syntaxe du langage hôte et ne nécessitent pas une formation particulière de la part de l'utilisateur. Pour le langage C++, on peut citer Compositional C++ qui est communément appelé CC++ [31].

Enfin, certains standards sont apparus sous la forme de langages de données parallèles dont les plus notoires ont été traités par Foster [57].

Dans les sections précédentes, l'accent a été mis sur l'importance de l'approche de conception et de programmation pour faire des programmes parallèles performants. Une fois ces phases terminées, ce sont les tests qui déterminent si telle ou telle approche est performante. La section qui suit explore justement les moyens de vérifier la qualité de la conception et de la programmation avec des outils et des formules quantitatives.

2.1.5 Les mesures de performance

Des mesures de performance de programmes parallèles sont mises à disposition pour justifier le choix d'une architecture, prédire le comportement d'un algorithme ou encore calculer le temps économisé par rapport à une autre version. La motivation derrière l'apparition de ces mesures de performance est la nécessité d'avoir des outils dans un domaine aussi complexe que le calcul parallèle. Scherr [135], considéré comme le pionnier de l'évaluation de performance parallèle, a insisté sur le fait que la réaction de la machine par rapport aux exigences de l'utilisateur a besoin d'être estimée à l'avance. À partir de ses recherches, beaucoup de métriques

ont vu le jour et certaines seront exposées dans la partie qui suit. Une fois appliquées, ces mesures sont autant de données statistiques qui alimentent des tables et graphiques pouvant servir de base à des théories et autres démonstrations.

2.1.5.1 Accélération

La première et probablement la plus importante mesure de performance d'un algorithme parallèle est l'accélération [9]. Elle est le rapport entre le temps d'exécution du meilleur algorithme connu sur 1 processeur et celui de la version parallèle. Sa formule générale est :

$$Accélération = \frac{Temps\ d'exécution\ séquentiel}{Temps\ d'exécution\ parallèle}. \tag{2.1}$$

Une des interprétations les plus connues de cette définition est la limite énoncée par Amdahl [13] plus connue sous le nom de "loi d'Amdahl" qui est la suivante : si s est la portion du code qui doit être exécutée séquentiellement, l'accélération est limitée par le haut par $\frac{1}{s+(1-s)/n}$ où n est le nombre de processeurs.

La loi d'Amdahl est connue comme étant la plus fondamentale et plus controversée des résultats dans le domaine de l'évaluation des performances parallèles [100]. Elle est considérée fondamentale par sa simplicité et sa généralité. En effet, elle définit une borne supérieure pour la performance d'un algorithme parallèle se basant sur un seul paramètre logiciel (la fraction séquentielle) et un seul paramètre matériel (le nombre de processeurs). La controverse vient du fait que cette borne impose des limites sévères aux performances et aux bénéfices d'avoir recours au parallélisme.

On distingue trois types d'accélérations : sous-linéaire (inférieure au nombre de processeurs), linéaire (égale au nombre de processeurs) et super-linéaire (supérieure au nombre de processeurs). Il faut toutefois noter que les accélérations sous-linéaires sont les plus répandues. Belding [20] et Lin [95] ont rapporté des accélérations super-linéaires mais la notion est toujours controversée. Des auteurs comme Faber *et al.* [51] stipulent qu'une telle accélération ne peut pas exister parce que les coûts de communications ne permettent pas de l'atteindre. D'un autre côté, des auteurs comme Parkinson [118] défendent l'accélération super-linéaire avec des

résultats expérimentaux.

2.1.5.2 Efficacité

Une autre métrique populaire est l'efficacité (formule 2.2). Elle donne une indication sur le taux d'utilisation des processeurs sollicités. Sa valeur est comprise entre 0 et 1 et peut être exprimée en pourcentage. Plus la valeur de l'efficacité est proche de 1, meilleures sont les performances. Une efficacité égale à 1 correspond à une accélération linéaire.

$$Efficacité = \frac{Accélération}{p} = \frac{Temps\ d'exécution\ Séquentiel}{Temps\ d'exécution\ Parallèle \times p} \tag{2.2}$$

Il existe d'autres variantes de cette métrique. Par exemple, "l'efficacité incrémentale" [9](Formule 2.3) qui donne l'amélioration du temps à l'ajout d'un processeur supplémentaire. Elle est aussi utilisée lorsque le temps d'exécution sur un processeur n'est pas connu. La formule de cette métrique a été généralisée (Formule 2.4) pour mesurer l'amélioration obtenue en augmentant le nombre de processeurs de n à m.

$$ie_m = \frac{(m-1) \cdot E[T_{m-1}]}{m \cdot E[T_m]}. \tag{2.3}$$

$$gie_{n,m} = \frac{n \cdot E[T_n]}{m \cdot E[T_m]}. \tag{2.4}$$

2.1.5.3 Autres mesures

Parmi les autres métriques utilisées pour mesurer les performances des algorithmes parallèles, on cite la "scaled speedup" (Accélération extensible) [9] (Formule 2.5) qui permet de mesurer l'utilisation de la mémoire disponible.

$$ss_m = \frac{Temps\ de\ résolution\ estimé\ pour\ un\ problème\ de\ taille\ n\ sur\ 1\ processeur}{Temps\ réel\ de\ résolution\ pour\ un\ problème\ de\ taille\ n\ sur\ m\ processeurs} \tag{2.5}$$

On cite aussi la "scaleup" (scalabilité) [9] (Formule 2.6) qui permet de mesurer l'aptitude du programme à augmenter sa performance lorsque le nombre de processeurs augmente.

$$su_{m,n} = \frac{Temps\ de\ résolution\ de\ k\ problèmes\ sur\ m\ processeurs}{Temps\ de\ résolution\ de\ nk\ problèmes\ sur\ nm\ processeurs} \tag{2.6}$$

Enfin, Karp et Flatt [88] proposent une autre métrique appelée la fraction séquentielle déterminée expérimentalement (Formule 2.7) et qui permet d'identifier les obstacles à de bonnes performances.

$$f_m = \frac{1/s_m - 1/m}{1 - 1/m} \tag{2.7}$$

Ces obstacles justement peuvent être de sources diverses et vont faire l'objet de la section suivante.

2.1.6 Facteurs influant sur la performance des algorithmes parallèles

La mesure de performances parallèles est une métrique complexe. Ceci est principalement dû au fait que les facteurs de performances parallèles sont dynamiques et distribués [100]. Le facteur communication est parmi les plus influents sur la performance de l'algorithme. Dans beaucoup de programmes parallèles, les tâches exécutées par les différents processeurs ont besoin d'accéder à des données communes (ex : multiplication matrice vecteur). Ceci crée un besoin de communication et freine la performance de l'algorithme. Ces communications sont d'autant plus importantes dans le cas où les processeurs auraient besoin de données générées par d'autres processeurs. Ces communications peuvent être minimisées en terme de volume de données et de fréquence d'échanges. Toutefois, le compromis à faire n'est pas toujours évident puisque les contraintes d'architecture entrent en compte. La taille des données échangées entre les processeurs peut également être limitée par des contraintes d'environnement. Par exemple, cette taille est fixée par défaut à 512Ko pour Mpich2 sur Infiniband [86].

Certains auteurs comme Grama *et al.* [73] expliquent les sources de ralentissement des algorithmes parallèles par les temps d'attente des processeurs. Une façon de combler ces temps d'attente est d'assigner des calculs indépendants aux processeurs inactifs.

Culler *et al.* [35] résument ces facteurs par la formule LogP (Latency, overhead, gap, Processor) qui identifie respectivement latence, surcharge, creux et processeurs. La latence est le délai dans le temps qui survient pendant la transmission d'un message d'un processeur à un autre. C'est la taille des données envoyées qui influence ce temps. Pour sa part, la surcharge est le temps requis pour l'initialisation d'un envoi ou d'une réception d'un message (hors données transmises) car, pendant ce temps, le processeur ne peut effectuer aucune autre opération. Ce paramètre est généralement fonction de l'architecture utilisée et peut être réduit en limitant la fréquence des échanges. Le creux est défini comme étant l'intervalle de temps minimum entre deux envois consécutifs ou deux réceptions consécutives sur un même processeur et représente une caractéristique du processeur (généralement un multiple du nombre de cycle par itération). Enfin, le nombre de processeurs impliqués est en lui-même un facteur de performance. Selon la taille du problème et la taille des données échangées, le nombre de processeurs va être plus ou bénéfique aux performances de l'algorithme.

Malony [100] stipule que la performance parallèle est difficile à mesurer, à identifier et à comprendre. Cependant, avec une approche scientifique basée sur l'hypothèse et l'expérimentation, on arrive à effectuer des améliorations. Cette approche est résumée dans la Figure 2.14.

2.1.7 Conclusion

Les notions qui ont été présentées dans ce chapitre parcourent de façon globale le parallélisme, la conception d'algorithmes, l'implémentation des programmes parallèles et les mesures de performances des programmes parallèles. Le parallélisme est un domaine très prometteur et a permis beaucoup d'avancées dans des secteurs exigeants en matière de calcul. Grâce à cette expansion et à la popularisation des machines parallèles, plusieurs domaines de l'informatique subissent des évolutions dans cette direction. Dans le domaine de la recherche opérationnelle, les métaheuristiques sont un type particulier d'algorithmes très intéressants à examiner du point de vue de la parallélisation. Dans la section suivante, nous allons définir plus spécifiquement leur fonctionnement.

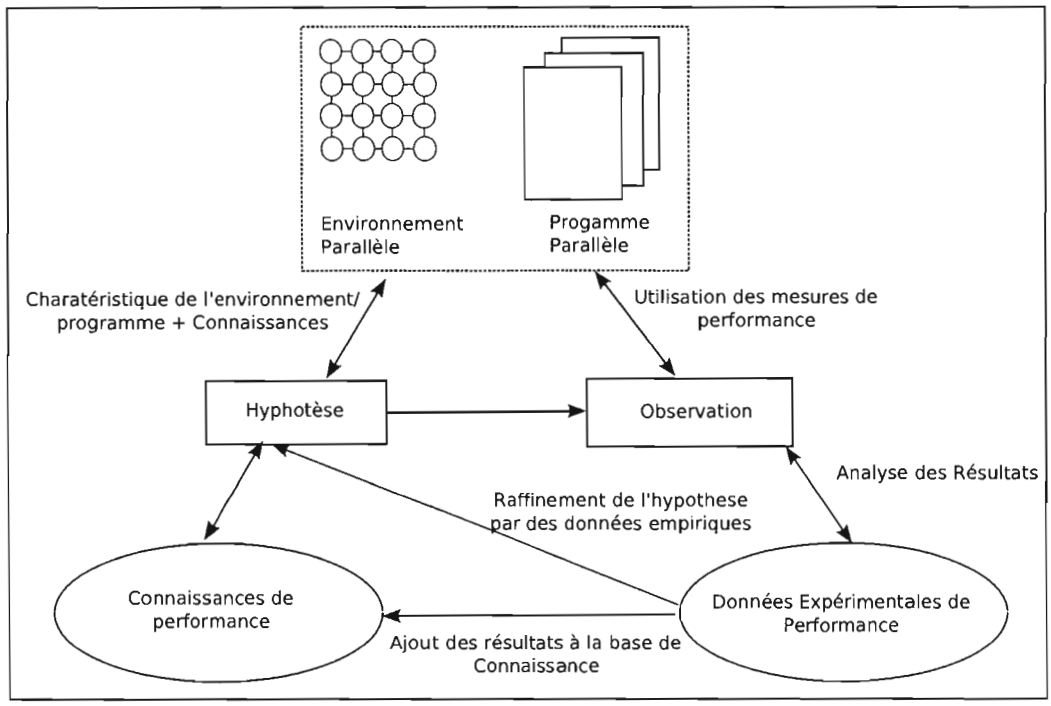

Figure 2.14 – Schéma général d'évaluation de performance parallèle [100]

2.2 Les métaheuristiques séquentielles et parallèles

2.2.1 Introduction

Trouver une solution à un problème d'optimisation combinatoire revient très souvent à concevoir un algorithme qui traite les données en entrée, effectue un traitement et produit une solution. L'algorithmique a cependant démontré que, pour certaines classes de problèmes, il n'existe pas d'algorithme capable de trouver la solution optimale. Une alternative consiste à utiliser une approche de solution de type métaheuristique (du grec meta : "au-delà", heuriskein : "trouver"). Dès lors, à défaut de trouver la solution optimale au problème, on effectue une démarche qui consiste à approximer cette solution. Les métaheuristiques sont utilisées dans les problèmes d'optimisation qui font partie de la classe de problèmes dits NP-Difficiles [62]. Dans cette section, les métaheuristiques les plus connues seront présentées ainsi que les versions parallèles issues de celles-ci.

2.2.2 Les algorithmes génétiques

2.2.2.1 Version séquentielle de l'algorithme génétique

Les algorithmes génétiques (AG) ont été introduits par les travaux de Holland [84] et Goldberg [69] et sont inspirés de la théorie de l'évolution et des sciences de la génétique. Ils appartiennent à la famille des algorithmes évolutionnaires [18]. Ils sont caractérisés par l'utilisation d'une population de structures multiples pour effectuer une recherche simultanée dans plusieurs zones de l'espace du problème. L'algorithme est guidé par un processus itératif qui, de génération en génération, conduit à une population plus adaptée. Le déroulement de ce processus est discuté plus en détails dans les sections qui suivent.

La métaheuristique débute son processus à partir d'un ensemble de solutions appelé Population où chacune de ces solutions est un Individu. La population de départ est généralement formée d'individus générés aléatoirement pour assurer la plus grande diversité possible. Un individu est, pour sa part, constitué d'une séquence de Gènes. Comme dans le processus de sélection naturelle, l'algorithme tente d'éliminer les mauvais traits des individus et de les faire évoluer afin d'en obtenir de meilleurs. Pour ce faire, l'algorithme comporte cinq phases principales : l'évaluation de la Fitness, la sélection, la reproduction, la mutation et le remplacement. Ces cinq étapes combinées font émerger de nouvelles générations à partir de la population courante. Ainsi, l'algorithme boucle sur ces quatre étapes et termine selon diverses conditions d'arrêt telles qu'une limite de temps, un nombre maximum de générations, un manque de diversité dans la population ou encore la stagnation de la fonction objectif depuis un certain nombre de générations. En résumé, la Figure 2.15 illustre un schéma d'un algorithme génétique de base. Dans cet exemple, une population P subit les opérateurs génétiques de l'algorithme au fil des générations notées T. On note par p un individu de cette population sur lequel sera appliqué l'opérateur de mutation.

L'évaluation de la Fitness est généralement représentée par une fonction f et est l'étape dans laquelle on mesure la qualité de chaque individu. Pour pouvoir décider de la qualité d'un individu et ainsi le comparer aux autres, il faut établir une mesure commune

```
Entrée  : Instance du problème
Sortie  : Une solution

T=0  ; Initialisation Population(T)  ; Evaluation(P(T))
Tant que (Condition d'arrêt non satisfaite)
        T=T+1
        Sélection(P(T))
        Reproduction(P(T))
        Mutation(p)
        Remplacement(P(T))
```

Figure 2.15 – Algorithme génétique de base [84]

d'évaluation. Par exemple, pour le problème du voyageur de commerce, on utilisera la distance totale comme fonction à minimiser tandis que dans le cas d'un problème d'ordonnancement, la minimisation du retard total pourrait être la fonction objectif.

La sélection consiste à choisir les individus les plus aptes à se reproduire. Il existe plusieurs mécanismes de sélection parmi lesquels : la sélection par tournoi, la sélection par roulette, la sélection par rang et la sélection déterministe. La sélection par tournoi consiste à choisir un nombre d'individus au hasard (taille du tournoi) et d'en sélectionner le meilleur selon la fonction objectif. Pour ce qui est de la roulette, on dispose au départ de l'ensemble des individus avec leurs "Fitness" respectives. Une probabilité est alors associée à chaque individu en fonction de sa "Fitness" et fera en sorte que les meilleurs individus ont plus de chances d'être choisis. Pour sa part, la sélection par rang trie les individus selon leurs "Fitness" et une nouvelle fonction f' leur est attribuée selon leur rang. Enfin, la sélection déterministe choisit directement les meilleurs individus d'une population. La sélection permet d'identifier un ensemble de couples qui pourront passer à l'étape de croisement.

Le croisement consiste à appliquer des procédures sur les individus sélectionnés pour donner naissance à un ou plusieurs (généralement deux) individus appelés "enfants", dont le code génétique est une combinaison de celui des parents. Les procédures d'échanges sont guidées par des opérateurs de croisements. Ces opérateurs sont un facteur très important pour la qualité des individus engendrés. Plusieurs études ont été conduites sur les opérateurs de croisement et démontrent que le choix doit se faire en fonction de la nature du problème [123] [83]. Dans les paragraphes qui suivent, les opérateurs ont été classés selon les caractéristiques qu'ils tentent de préserver. On peut ainsi distinguer trois catégories d'opérateurs [107] : ceux qui préservent

la position absolue, ceux qui préservent l'ordre et ceux qui préservent l'adjacence (les arcs). À noter que les opérateurs cités s'appliquent lorsque la codification des gènes est telle que chaque gène est unique comme c'est le cas dans le problème du voyageur de commerce. On s'intéresse à cette codification et à ces opérateurs en particulier parce qu'ils s'appliquent au problème traité dans le chapitre suivant.

Parmi les opérateurs qui préservent la position absolue, on note :

- PMX : Partially Mapped Crossover [68]

  PMX est un opérateur de croisement à deux points de coupure qui définit un segment de même longueur dans chacun des parents *P1* et *P2*. Les segments sont indiqués avec une trame foncée dans Figure 2.16 partie (a). Ces segments sont copiés chez les enfants opposés *E1* et E2. Par exemple, *E1* a hérité du segment de *P2* (b). Avec les gènes de ces segments, on établit une liste de correspondance. Cette liste va servir à placer les gènes redondants et elle est formée de la manière suivante : pour chaque position du segment on note x le gène qui s'y trouve et y celui de l'autre enfant dans la même position. Tant que y est retrouvé ailleurs dans le segment de départ, on note y' son correspondant dans l'autre enfant et on remplace y par y'. Par exemple, le gène correspondant à "1" de *E1* est "6" mais ce gène existe aussi dans *E1* et son correspondant est "3". Ainsi dans la liste de correspondance, on note que "1" a pour correspondant "3". La liste complètement formée se trouve à la marge de la partie (b) du schéma. On procède ensuite au placement des gènes hors segment en les copiant des parents respectifs. Par exemple, copier "1" de *P1* dans *E1* provoque un conflit puisque que ce gène existe déjà. On utilise alors son correspondant dans la liste qui est "3". En procédant de manière itérative, on arrive à former les enfants *E1* et *E2* illustrés à la partie (c) de la Figure 2.16.

- CX : Cycle Crossover [115]

  CX est un opérateur qui satisfait la condition suivante : chaque gène d'un enfant provient de l'un des parents à la même position. Les enfants sont donc formés en copiant un gène d'un parent et en éliminant l'autre à la même position puisqu'il va

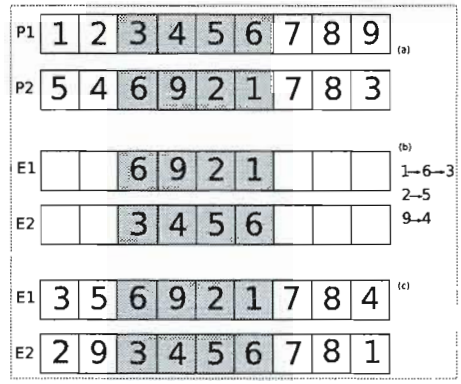

Figure 2.16 – Opérateur de croisement PMX [68]

appartenir au deuxième enfant. Une fois que les positions occupées sont copiées par élimination, on a complété un cycle. Les places restantes des deux enfants sont complétées par les parents opposés. Dans l'exemple présenté à la Figure 2.17 partie (a), la première position de *E1* est attribuée à "1" provenant de *P1*. Le gène de *P2* situé à la même position "4" est recherché dans *P1* et se retrouve se trouve à la quatrième position. Le gène "4" est copié dans *E1* et son correspondant "8" est recherché dans *P1* et retrouvé à la huitième position. "8" est copié dans *E1* à cette même position et son correspondant "3" est recherché dans *P1* et retrouvé à la troisième position. "3" est copié dans *E1* et son correspondant "2" est recherché dans *P1* et retrouvé à la deuxième position. "2" est copié et son correspondant "1" est retrouvé dans *P1* à la première position or cette position est déjà occupée dans *E1* donc le cycle est terminé. Le cycle de *E1* est "1,4,8,3,2" et de manière analogue, le cycle "4,1,2,3,8" est formé pour E2. En dernier lieu, il faut combler les positions vacantes à partir des parents opposés. Par exemple, les gènes "7,6,9" de *E1* proviennent de *P2*. On obtient ainsi les enfants illustrés dans la partie (b) de la Figure 2.17.

- Échange de séquence (Subtour Exchange) [155]

  Cet opérateur sélectionne des sous-ensembles non identiques contenant des gènes communs aux deux parents et les échange. Comme illustré à la Figure 2.18, les sous-ensembles "4567" et "5647" respectivement de *P1* et de *P2* sont sélectionnés dans la partie (a). Dans la partie (b), ces sous-ensembles sont permutés pour

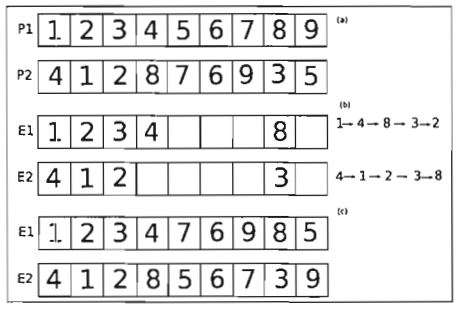

Figure 2.17 – Opérateur de croisement CX [115]

former les enfants *E1* et E2. Comme les deux sous-ensembles sont de même taille et contiennent les mêmes gènes, il ne peut pas y avoir de conflit après l'échange.

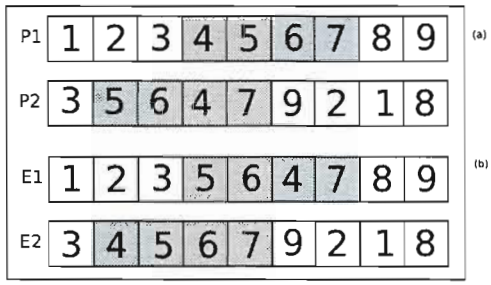

Figure 2.18 – Opérateur échange de séquence [155]

- Croisement uniforme de permutation [39]

  Le croisement uniforme de permutation génère un masque binaire aléatoire. C'est une séquence binaire de même longueur que l'individu telle que les gènes marqués "0" sont conservés dans un des parents et ceux marqués "1" dans l'autre. Les gènes non marqués du premier parent sont permutés selon l'ordre dans lequel ils apparaissent dans le deuxième parent (et réciproquement pour le parent 2). L'exemple de la Figure 2.19 présente un masque à neuf éléments dans la partie (a). Ce masque appliqué aux parents *P1* et *P2* donne la sélection représentée dans la partie (b). En dernier lieu, on procède au réarrangement des gènes sélectionnés. Par exemple, les gènes "3-4-6-8" sont marqués dans *P1* et apparaissent dans *P2* dans l'ordre suivant : "8-6-4-3". C'est donc dans cet ordre qu'ils seront réarrangés dans *E1* comme le montre la partie (c) de la Figure 2.19.

Parmi les opérateurs qui préservent l'ordre, on note :

- OX : Order Crossover [38] [115]

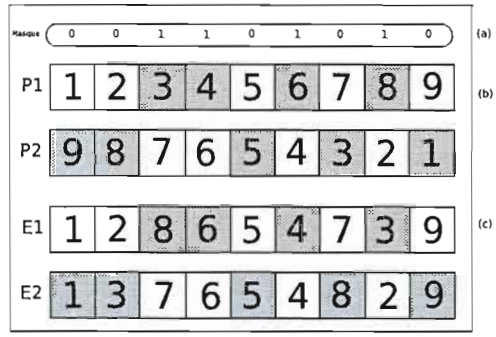

Figure 2.19 – Opérateur uniforme de permutation [39]

OX est un opérateur avec deux points de coupure qui copie le segment formé par ces deux points de coupure dans les deux enfants. Par la suite, il recopie, à partir du deuxième point de coupure ce qui reste des gènes dans l'ordre du parent opposé en évitant les doublons. La Figure 2.20 présente un exemple du déroulement du croisement avec l'opérateur OX. Dans la partie (a), un segment est formé par les points de coupure dans les deux parents et ces segments sont copiés tels quels dans les enfants *E1* et *E2* comme le montre la partie (b). Enfin, on procède à la copie des gènes situés hors du segment copié. Pour cela, on se place à partir du deuxième point de coupure et on choisit les gènes non redondants provenant du parent opposé. Par exemple, dans la partie (c) de la Figure 2.20, on essaie de placer le gène "6" de *P2* après le deuxième point de coupure dans *E1* mais ce gène existe déjà à l'intérieur du segment. Il est donc ignoré et on passe au suivant. Le gène "2" ne présente pas de conflit, il est donc copié et ainsi de suite jusqu'à former les deux enfants *E1* et *E2* tel qu'illustré à la Figure 2.20 partie (c).

- LOX : Low Order Crossover [115]

  LOX est un opérateur semblable à OX à la différence que la distribution hors segment se fait à partir du début du parent opposé et non pas à partir du deuxième point de coupure. Comme illustré à la Figure 2.21 partie (a), un segment est formé par deux points de coupure chez les deux parents *P1* et *P2* et ces segments sont copiés respectivement dans *E1* et *E2* comme montré à la Figure 2.21 partie (b). Enfin, dans la partie (c), les gènes hors segment sont copiés de manière similaire à

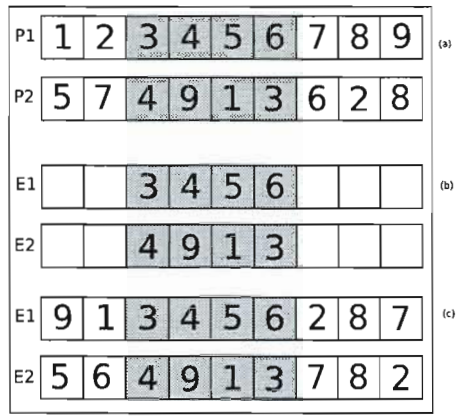

Figure 2.20 – Opérateur de croisement OX [38] [115]

l'opérateur OX, à la différence que le point de départ est la première position du parent opposé. Par exemple, on commence au deuxième point de coupure de *E1* et on copie le premier gène de *P2*. Le gène "5" pose conflit et est ignoré pour passer au suivant. Le deuxième gène, "7", ne pose pas de conflit et il est alors copié. On procède ainsi jusqu'à former les enfants *E1* et *E2* tel qu'illustré à la partie (c) de la Figure 2.21.

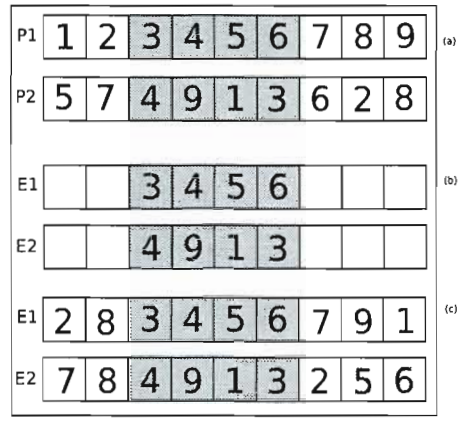

Figure 2.21 – Opérateur de croisement LOX [115]

- OBX : Order Based Crossover [142]

OBX est un opérateur qui s'intéresse à l'ordre relatif des gènes dans les parents. Un sous-ensemble de gènes est sélectionné dans le premier parent. L'ordre des gènes de ce sous-ensemble va être imposé au premier enfant. Les positions auxquelles ces gènes vont être affectés sont retrouvées à partir du deuxième parent. La Figure 2.22 illustre le fonctionnement de l'opérateur OBX. Dans la partie (a), un segment

est formé en sélectionnant un sous-ensemble chez les deux parents *P1* et *P2*. Les gènes de ces segments sont copiés respectivement dans *E1* et *E2* dans les positions auxquelles ils apparaissent dans le parent opposé. Par exemple, la partie (b) de la Figure 2.22 montre la transformation subie par les gènes sélectionnés dans la partie (a). En effet, les gènes sélectionnés de *P1* sont "3-4-5-6" et apparaissent dans *P2* aux positions 1,3,6 et 7. Ce sont donc dans ces positions qu'on va les retrouver dans E1. La même démarche est suivie pour E2. Enfin, les positions vacantes sont remplies à partir des parents opposés. Comme le montre la partie (c) de la Figure 2.22, les positions vacantes de *E1* proviennent de *P2* et inversement pour E2.

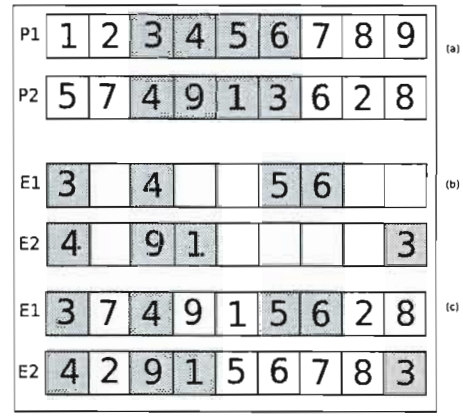

Figure 2.22 – Opérateur de croisement OBX

– PBX : Position Based Crossover [142]

Dans cet opérateur, les enfants sont formés à partir d'un sous-ensemble de positions issues d'un parent et les gènes correspondants à ces positions de l'autre parent. Les autres positions sont remplies avec les gènes restants dans le même ordre relatif que le deuxième parent. Le fonctionnement de l'opérateur PBX est illustré à la Figure 2.23. Dans la partie (a), des positions sont choisies au hasard dans les deux parents *P1* et *P2* et ces positions seront copiées respectivement dans les enfants *E1* et *E2* comme illustré dans la partie (b) de la Figure. Enfin, les positions vacantes sont remplies selon l'ordre relatif du parent inverse. Dans E1, il reste à placer "2,4,6,7,9" et ils apparaissent dans *P2* dans l'ordre suivant "7-4-9-6-2". C'est donc dans cet ordre qu'ils sont placés dans E1. Le même procédé est suivi pour *E2* et les enfants

sont formés comme le montre la partie (c) de la Figure 2.23.

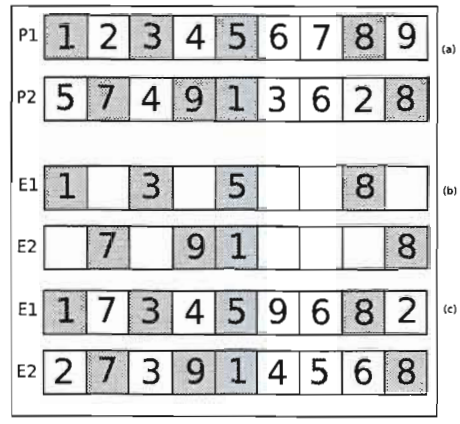

Figure 2.23 – Opérateur de croisement PBX

Parmi les opérateurs qui préservent l'adjacence, on note :

- Croisement par recombinaison d'adjacences (Edge Recombinaison Crossover) [154]

  Cet opérateur tente de faire hériter des adjacences existant dans les deux parents. Il utilise une structure de données spéciale appelée "carte d'adjacence" (edge map). Cette carte évolue à chaque fois qu'un gène est sélectionné pour renseigner sur les adjacences à chaque étape. La Figure 2.24 montre le fonctionnement de l'opérateur par recombinaison d'adjacences. La partie (a) montre la liste des adjacences qu'on peut extraire de la représentation des parents *P1* et *P2*. L'enfant E1, représenté dans la partie (b), est formé en choisissant une première position de l'un des parents (celle de *P1* a été choisie dans l'exemple). Le gène "1" est éliminé de la liste d'adjacence et on choisit un de ses adjacents qui possède le minimum de voisins. Dans l'exemple, c'est le "7" qui est choisi et on réitère le procédé jusqu'à former l'enfant *E1* en entier.

- Croisement heuristique (Heuristic Crossover) [75]

  Cet opérateur est considéré comme une classe à part entière. Il fonctionne en sélectionnant un gène au hasard et quatre de ses adjacences (deux de chaque parent). Il compare ainsi les quatre adjacences et sélectionne la plus courte (si la matrice des distances est connue dans le probleme) ou la moins coûteuse (en recalculant la fonction objectif). L'opérateur boucle sur cette étape jusqu'à ce que l'enfant soit entièrement formé. Dans l'exemple de la Figure 2.25, le gène "3" est choisi de manière

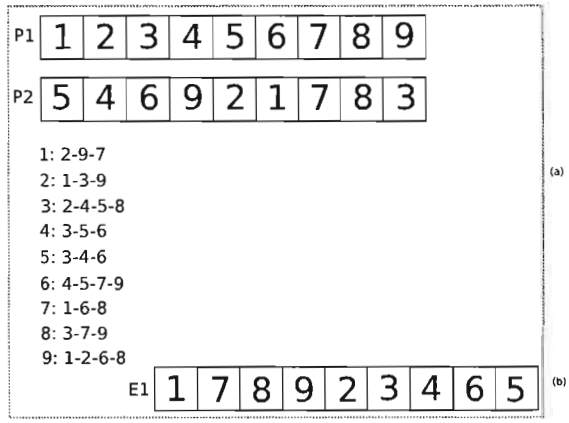

Figure 2.24 – Opérateur de croisement par recombinaison d'adjacences

aléatoire et tous ses voisins dans les deux parents *P1* et *P2* sont comparés. Le choix du gène suivant repose sur les coûts de passage ou encore sur la fonction objectif.

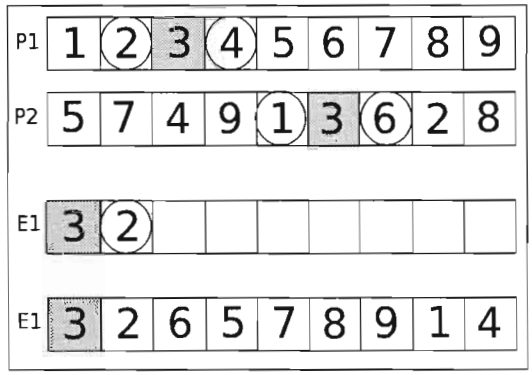

Figure 2.25 – Opérateur de croisement heuristique

Le troisième opérateur génétique d'un algorithme génétique est la mutation. C'est un processus où un changement mineur de code génétique est appliqué à un individu pour introduire de la diversité et ainsi éviter de tomber dans des optimums locaux. Le paramètre associé à cet opérateur est la probabilité de mutation qui exprime en pourcentage, les chances pour que la mutation soit déclenchée. La mutation possède également des opérateurs dont on peut citer les suivants :

- Mutation par inversion : Deux positions sont sélectionnées au hasard et tous les gènes situés entre ces positions sont inversés. La Figure 2.26 montre un exemple de mutation par inversion. L'individu i avant mutation est représenté dans la partie (a) avec le segment formé par les deux positions sélectionnées. L'inversion est effectuée

à l'étape (b) afin de produire l'individu *i'*.

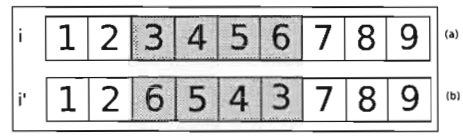

Figure 2.26 – Mutation par inversion

- Mutation par insertion : Deux positions sont sélectionnées au hasard et le gène appartenant à l'une est inséré à l'autre position. Dans la Figure 2.27partie (a), les gènes "3" et "6" de l'individu *i* sont sélectionnés. La deuxième étape, illustrée par la partie (b), montre l'insertion de "6" avant "3" et le décalage de tous les autres gènes.

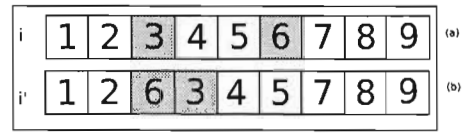

Figure 2.27 – Mutation par insertion

- Mutation par déplacement : Une séquence est sélectionnée au hasard et déplacée vers une position elle-même tirée au hasard. Un exemple de mutation par déplacement est illustré à la Figure 2.28. Dans la partie (a), la séquence "3-4-5-6" est sélectionnée et déplacée à la dernière position pour former l'individu *i'* représenté dans la partie (b).

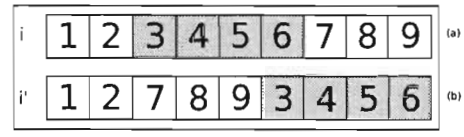

Figure 2.28 – Mutation par déplacement

- Mutation par permutation : Deux positions sont sélectionnées au hasard et les gènes qui s'y trouvent sont permutés. Comme illustré à la Figure 2.29, les éléments "3" et "6" sont sélectionnés dans *i* (partie (a)) et permutés dans *i'* (partie (b)).

Certains auteurs tels que Rubin et Ragatz [134] ont implémenté une version particulière de la mutation contenant de la recherche locale. La recherche locale se base sur le concept de voisinage. Ce mécanisme remplace la solution actuelle par une meilleure dans son voisinage immédiat si celle-ci existe. Appliquée à la mutation, la recherche locale va permuter les gènes voisins de l'individu en tentant de l'améliorer.

Dans le même esprit, on peut citer la mutation heuristique (Heuristic Mutation) qui

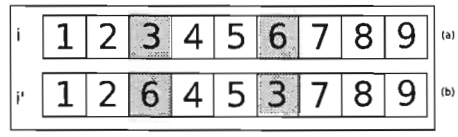

Figure 2.29 – Mutation par permutation

est schématisée à la Figure 2.30. Cette mutation consiste à sélectionner un nombre aléatoire de positions, faire toutes les permutations possibles des positions sélectionnées, évaluer les solutions et en sélectionner la meilleure. Ces deux dernières versions de la mutation sont très

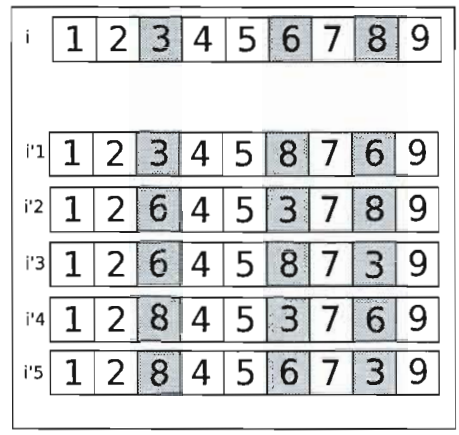

Figure 2.30 – Mutation heuristique

coûteuses en terme de temps d'exécution et ne sont pas fidèles au concept génétique de la mutation puisqu'elle est supposée être aléatoire et minime.

Enfin, le dernier opérateur génétique décrit dans l'algorithme est le remplacement. Ce mécanisme permet la recombinaison entre les nouveaux individus créés et ceux qui se trouvent présentement dans la population. Plusieurs stratégies de remplacement existent dans la littérature [63] parmi lesquelles on cite : l'élitisme, le remplacement complet, le remplacement partiel et le remplacement aléatoire.

L'élitisme consiste à recombiner les parents et les enfants en éliminant ceux qui donnent les moins bons résultats selon la fonction objectif. Le remplacement complet fait en sorte que la nouvelle population soit uniquement constituée d'enfants. Ceci peut se faire à condition que le nombre d'enfants soit égal à la taille de population. Le remplacement partiel quant à lui, reconstruit la nouvelle population avec une proportion d'individus issue des parents et une autre issue des enfants. Enfin, le remplacement aléatoire sélectionne au hasard les

individus à conserver sans pour autant éliminer les meilleurs individus.

Les algorithmes génétiques ont été appliqués avec succès pour la résolution de plusieurs problèmes d'optimisation combinatoire. Une revue de ces problèmes a été faite par Michalewicz [107].

2.2.2.2 Version parallèle de l'algorithme génétique

Les algorithmes génétiques s'adaptent très bien par leur structure à l'exécution en parallèle. Avec la popularisation des machines parallèles, des versions parallèles de l'algorithme génétique ont vu le jour et ont fait l'objet de plusieurs études [29]. Luque *et al.* [98] ont retracé les importants travaux qui ont marqué le domaine des algorithmes génétiques parallèles. Ceux-ci sont résumés dans le Tableau 2.1.

Algorithme	Référence	Description
ASPARAGOS	[72]	AG hautement parallèle. Applique le "hill-climbing" si il n'y a pas d'amélioration
DGA	[148]	AG à population distribuée
GENITOR II	[153]	Modèle en îlot
ECO - GA	[37]	AG hautement parallèle
PGA	[113]	AG en îlot avec migration du meilleur individu et "hill climbing" local
SGA-Cube	[50]	Modèle en îlot implémenté sur nCUBE 2
EnGENEer	[133]	Parallélisation globale
PARAGENESIS	[138]	Modèle en îlot implémenté pour le CM-200
GAME	[138]	Ensemble d'outils orientés objet pour programmation générale
PEGAsuS	[132]	Modèle permettant une programmation de haut niveau sur des machines de type MIMD
DGENESIS	[103]	Modèle en îlot avec migration
GAMAS	[122]	Modèle en îlot avec 4 noeud
iiGA	[95]	AG en îlot avec injection d'individus, hétérogène et asynchrone
PGAPack	[94]	Parallélisation globale de la fonction objectif
CoPDEB	[2]	Modèle en îlot avec un opérateur différent par noeud
GALOPPS	[71]	Modèle en îlot
MARS	[144]	Environnement parallèle avec tolérance
RPL2	[126]	Modèle en îlot très flexible, permettant de définir de nouveaux modèles
GDGA	[82]	Modèle en îlot ayant une topologie en hypercube
DREAM	[15]	Framework pour algorithmes évolutionnaires distribués
MALLBA	[7]	Framework pour algorithmes parallèles
Hy4	[8]	Modèle en îlot hétérogène ayant une topologie en hypercube
ParadisEO	[26]	Framework pour algorithmes parallèles

Tableau 2.1 – Travaux en algorithmes génétiques parallèles [98]

Parmi les travaux les plus récents sur les algorithmes génétiques parallèles, on note ceux de Guo *et al.* [77] qui ont utilisé cette métaheuristique pour la résolution du problème de "conduction de chaleur inverse" (Inverse Heat Conduction) avec des résultats très encourageants.

On cite également Yeh *et al.* [156] qui ont proposé deux algorithmes génétiques parallèles pour la gestion de configuration de produit (Product Configuration Management).

Après avoir présenté sommairement quelques travaux dans le domaine des algorithmes génétiques parallèles, on va aborder plus en détails les caractéristiques de ces algorithmes, à savoir les modèles et les topologies.

Il existe principalement deux modèles d'algorithmes génétiques parallèles : le modèle maître-esclave et le modèle en îlot. Dans le premier modèle, un processeur est dédié aux étapes de sélection, de croisement, de mutation et de remplacement et le reste des processeurs s'occupe de calculer la fitness des individus. Ce modèle possède plusieurs avantages dont la facilité d'implémentation et d'exploration du même espace de recherche que la version séquentielle et ce, plus rapidement. Parmi les principaux travaux sur ce modèle, on peut citer Bethke [22] et Grefenstette [74] qui en ont vanté les mérites. D'autres recherches ont identifié plus tard des problèmes d'extensibilité [54], [80] et [1]. En effet, plus le nombre de processeurs utilisé est grand, plus l'efficacité de l'algorithme diminue à cause de la surcharge de communication.

Ensuite, le modèle en îlots ou encore modèle distribué, est le plus populaire parmi les algorithmes génétiques parallèles. Ce modèle a été initialement introduit par les travaux de Grosso [76]. L'objectif de cette recherche était d'étudier l'interaction entre plusieurs sous-composantes parallèles d'entités évolutives. Grosso a constaté que les solutions trouvées par les îlots isolés étaient de loin moins bonnes que celles atteintes avec une seule grande population. Cependant, lorsque les migrations sont fréquentes, la qualité de solutions est équivalente dans les deux versions. Ceci amène une caractéristique très importante du modèle en îlot : la migration. En effet, pour faciliter le processus de sélection et pour donner de meilleurs résultats, des individus sélectionnés parmi une sous-population peuvent être déplacés vers une autre. Le choix des individus candidats à la migration peut être paramétrable mais, très souvent, il s'agit de faire écouler une période de temps pendant laquelle aucune migration n'est permise pour ensuite choisir un individu (le meilleur, le pire ou aléatoire) et le substituer avec un autre (le meilleur, le pire ou aléatoire) d'une population voisine. La migration est régie par deux paramètres : l'intervalle de migration et le taux de migration. L'intervalle de migration

est la durée après laquelle on permet les échanges entre les sous-populations. Plus il est faible, plus les migrations sont fréquentes. Dans les travaux de Petty *et al.* [119], une copie du meilleur individu est envoyée à toutes les sous-populations après chaque génération dans le but d'assurer un bon mélange des individus. Les conclusions de cet essai sont qu'un tel niveau élevé de communication donne des résultas de même qualité qu'un algorithme génétique séquentiel avec une population sans restriction de croisement. Le deuxième paramètre à considérer dans la migration est le taux. C'est le pourcentage de la population qui sera copiée vers la sous-population de destination. Les recherches de Grosso [76] ont montré qu'il existait un taux de migration critique au-dessous duquel les performances de l'algorithme sont freinées à cause de l'isolation des sous-populations, et au delà duquel les solutions trouvées sont de même qualité que la version séquentielle.

La deuxième caractéristique des algorithmes génétiques parallèles qui va être abordée est la topologie. Il s'agit de la manière par laquelle les îlots sont reliés et ceci a une grande influence sur la propagation des bons individus. En effet, dans le cas extrême où aucune communication entre les îlots n'est permise, les sous-populations n'ont aucune influence les unes sur les autres. Dans le cas contraire où un maximum de connexions est établi, on converge trop vite vers un minimum local et la diversité est trop faible pour permettre à une bonne solution d'émerger.

Plusieurs topologies ont été identifiées et testées [28]. L'efficacité d'une topologie revient à analyser le graphe qu'elle forme. Si l'algorithme n'est exécuté que sur deux sous-populations, la topologie n'a aucune influence vu que seuls les voisins directs importent. Cependant, si plus que deux sous-populations sont sollicitées, le facteur connectivité du graphe intervient. En effet, Cantù-Paz [30] a mis en évidence une relation directe entre la connectivité du graphe formé par ces sous-populations et la qualité des solutions de l'algorithme.

Les algorithmes génétiques sont le thème principal de ce mémoire. Néanmoins, d'autres métaheuristiques séquentielles et parallèles ont été développées et vont être citées dans la section qui suit.

2.2.3 Autres métaheuristiques

2.2.3.1 Optimisation par colonie de fourmis

L'optimisation par colonie de fourmis (OCF) [44] est une approche métaheuristique issue de l'étude de comportement des fourmis. L'analogie vient du fait que pendant l'approvisionnement de la colonie, les fourmis laissent des traces de phéromone qui marquent leur passage sur un chemin donné. La phéromone est une hormone très éphémère et reconnaissable entre fourmis d'une même colonie. Quand la nourriture est trouvée, le chemin qui y a mené sera plus imbibé en phéromone que les autres. Progressivement, toutes les fourmis seront au courant de ce chemin et tenteront de le rendre encore plus court.

Dans un algorithme d'OCF, un problème donné est ramené à un graphe. Des fourmis virtuelles parcourent ce graphe et effectuent un traitement probabiliste sur une solution donnée. Le modèle probabiliste utilisé est appelé Modèle Phéromone (Pheromone Model) et consiste en un ensemble de paramètre dont les valeurs sont appelées Valeurs Phéromones (Pheromone Values) [46]. En utilisant ces valeurs, la métaheuristique construit un nouveau segment et évalue la modification par rapport à ce qui a été fait précédemment. Si cette modification est bonne, la trace laissée va être de plus en plus imbibée, attirant de ce fait d'autres fourmis qui vont à leur tour l'étudier. La trace abandonnée va être de moins en moins attirante pour les fourmis et va progressivement disparaître.

Les versions de l'OCF se distinguent par la manière dont les valeurs phéromones sont mises à jour et les principales sont Ant Colony System (ACS) [47] [48] et Max-Min Ant System (MMAS) [140]. Une revue complète des algorithmes OCF [45] ainsi que leurs applications [44] a été faite par Dorigo. Le schéma de base d'un algorithme d'optimisation par colonie de fourmis est illustré à la Figure 2.31. L'algorithme commence par initialiser sa matrice de phéromones avant de débuter dans une première boucle pour un nombre cycle défini par la variable *NbCycles*. Une deuxième boucle est imbriquée à celle-ci pour générer les fourmis de la colonie. Une fourmi est représentée par la variable k et le nombre total de fourmis par *NbFourmis*. Chaque fourmi k construit sa solution s en mettant à jour sa trace

de phéromone locale et évalue ensuite la Fitness F lié à s. Une fois toutes les fourmis créés, l'algorithme identifie la meilleure solution du cycle s^{bc} avec sa valeur F^{bc} et utilise ces valeurs pour une mise à jour globale de la phéromone et de la meilleure solution connue s^{bg} ainsi que sa valeur F^{bg}.

```
Initialiser Phéromone
Pour (Nombre de cycles de 1 à NbCycles)
        Pour (k=1 jusqu'à NbFourmis)
                Construire une solution s
                Mise à jour locale de la Phéromone
                Evaluation de F pour s
        Définir la meilleure solution du cycle courant s^bc et sa valeur F^bc
        Mise à jour globale de la Phéromone avec s^bc
        Mise à jour de la meilleure solution connue s^bg et sa valeur F^bg
```

Figure 2.31 – Optimisation par colonies de fourmis ACS [59]

La nature de l'algorithme d'OCF rend sa parallélisation naturelle, que ce soit dans les données ou dans le domaine. Plusieurs modèles d'OCF parallèles peuvent être retrouvés dans la littérature, nous citons les plus marquants.

Stutzle [139] a abordé la parallélisation de l'OCF avec des exécutions indépendantes. Pour leur part, Michel *et al.* [108], Mendes *et al.* [104] et Middendorf [109] ont proposé des modèles distribués (en îles) avec diverses stratégies d'échanges entre colonies. D'autre auteurs comme Talbi *et al.* [145] et Rahoual *et al.* [129] ont implémenté des modèles maître-esclave où le processeur maître distribue les tâches aux esclaves.

Plus récemment, les bibliothèques standards de programmation parallèle ont été utilisées pour l'implémentation de modèles OCF parallèles. Nous citons Randall *et al.* [130] et Manfrin *et al.* [101] qui ont proposé une parallélisation de l'OCF en ayant recours au standard MPI. Delisle *et al.* [42] ont fait de même en utilisant OpenMP.

2.2.3.2 Recuit simulé

Le recuit simulé (RS) est une métaheuristique connue pour être la plus ancienne [24]. Elle a été proposée par Kirkpatrick en 1983 [89] d'après le travail de Metropolis [106] d'où elle puise ses origines statistiques. Le terme recuit est inspiré d'un processus utilisé en métallurgie dans lequel on fait alterner les cycles de chauffage et de refroidissement des métaux pour

minimiser l'énergie des matériaux. Cette métaheuristique utilise une approche de Monte Carlo [24] pour simuler le comportement d'un système qui tend à atteindre un équilibre thermal et ainsi devenir stable. Cette analogie est utilisée pour résoudre les problèmes d'optimisation combinatoire. Il a été prouvé qu'en surveillant de manière précise le taux de refroidissement, l'algorithme est capable de trouver l'optimum global mais paradoxalement, ceci prendrait un temps infini. C'est pour cela que d'autres versions du recuit simulé, le Fast annealing et le Very Fast Simulated Reannealing (VFSR) qui sont exponentiellement plus rapides que la version de base, sont utilisées pour surmonter le problème des délais. Le recuit simulé possède un sérieux avantage par rapport aux autres méthodes par le fait qu'il ne se fait par piéger dans les minima locaux [25].

Le recuit simulé est cependant complexe à paramétrer. Pour ce qui est des problèmes NP-Difficiles, celui qui a été le plus étudié est sûrement celui du voyageur de commerce. Le fonctionnement d'un algorithme de recuit simulé de base est énoncé dans la Figure 2.32. Dans cet algorithme, une solution est notée s et est initialement générée de manière aléatoire grâce à la fonction *Generer_Solution_Aleatoire*. La température T est aussi initialisée à une valeur initiale T_0. L'algorithme explore par la suite le voisinage immédiat de la solution noté N et compare les deux valeurs s et s' selon la fonction objectif f. Si s' est une meilleure solution que s, elle la remplace. Sinon, la nouvelle solution est acceptée selon une fonction de probabilité notée $p(T,s',s)$. Enfin l'algorithme met à jour la valeur de la température de l'itération en cours.

```
s= Generer_Solution_Initiale()
T=T0
Tant que (condition d'arrêt non satisfaite)
        s'=Selection_Aleatoire(N(s))
        si f(s')<f(s)
                s=s'
        sinon
                Accepter s' en tant que nouvelle solution avec une probabilité p(T,s',s)
        Mise_A_Jour(T)
```

Figure 2.32 – Algorithme de recuit simulé [89]

Cette métaheuristique est l'une des premières basées sur la recherche locale à être parallélisée. Les différents modèles théoriques de parallélisation ont été largement couvert par

Azencott [17]. D'un autre coté, Aydin *et al.* [16] ont présenté une revue des travaux les plus marquants dans le domaine du Recuit simulé parallèle. Ainsi, nous allons citer les travaux qui sont apparus après cette revue de la littérature. Les auteurs cités précédemment ont identifié des versions classiques de recuit simulé parallèle. Les travaux qui ont suivi ont pris une autre direction en essayant de faire des versions hybrides. A titre d'exemple, Debudaj-Grabysz *et al.* [41] ont fait combiner MPI et OpenMP pour mettre en place une nouvelle version de recuit simulé parallèle avec un modèle hybride de communication. D'autre part, Wang *et al.* [152] ont fait une version hybride en combinant les algorithmes génétiques avec le recuit simulé, cette version est appelée Recuit Simulé Génétique Parallèle (Parallel Genetic Simulated Annealing PGSA).

Dernièrement, Chen *et al.* [32] ont étudié quatre approches différentes de parallélisation du recuit simulé dont deux en y incorporant des algorithmes génétiques. Ils ont conclu que l'hybridation par les algorithmes génétiques est une voie prometteuse pour les versions parallèles du recuit simulé.

2.2.3.3 GRASP

Le GRASP (Greedy Randomized Adaptive Search Procedure) [52] [120] est une métaheuristique qui consiste en une suite de solutions construites par une approche vorace et à leur optimisation par l'exploration de leurs voisinages respectifs.

Chaque itération de cette métaheuristique consiste en une phase de construction de solution et en une exploration de voisinage. La construction se fait élément par élément, le choix du prochain élément se faisant par une approche vorace. GRASP essaie de tirer à la fois avantage de l'approche vorace et de l'approche aléatoire. GRASP garde une trace de la meilleure solution et la retourne à la fin de l'algorithme. Cette heuristique est appelée dynamique, en opposition aux autres heuristiques statiques qui assignent un score aux éléments seulement avant la construction. Par exemple, pour le problème du voyageur de commerce qui est basé sur les coûts des arcs dans le graphe, plus l'arc est court, plus son score est haut. La deuxième phase de l'algorithme est une recherche locale qui peut être basique ou encore faire appel à des techniques plus avancées telles que le recuit simulé vu à la Section

2.2.3.2. Le GRASP a l'avantage d'être facile à implémenter avec très peu de paramètres et d'avoir plusieurs applications possibles. La Figure 2.33 résume le fonctionnement du GRASP. L'algorithme de GRASP initialise la meilleure solution s^* à l'infini. Il effectue par la suite une construction vorace sur la solution en cours et trouve un minimum local dans le voisinage noté N. Si cette solution est meilleure que la solution en cours selon la fonction f elle est gardée.

```
s* = ∞
Tant que (Condition d'arrêt non satisfaite)
        Construire_Solution_Vorace(s)
        Trouver le minimum local s' dans N(s)
        si f(s')<f(s*)
                s*=s'
Retourner meilleure solution de s*
```

Figure 2.33 – Algorithme GRASP [52]

Deux stratégies de parallélisation de GRASP ont été identifiées par Alvim [40]. La première consiste à décomposer l'espace de recherche en régions et à appliquer GRASP à chacune d'elle en parallèle. La deuxième consiste à partitionner les itérations et à les distribuer parmi les processeurs. La majorité des algorithmes qui utilisent ces deux stratégies tombent dans la catégorie Marche Multiple Thread Indépendant (Multiple Walk Independent Thread). Parmi les travaux effectués en adoptant cette stratégie, on cite Padalos *et al.* [117], Martins *et al.* [102]. L'apparition plus tard de versions hybrides de GRASP parallèles avec "Path Relinking" a fait naître une nouvelle catégorie appelée Marche Multiple Thread Coopératif (Multiple Walk Cooperative Thread). Aiex *et al.* [5] [6] ont utilisé cette technique pour la résolution du problème d'ordonnancement job shop et du "Three Index Assignement Problem" avec le standard MPI. Les versions hybrides gagnent de plus en plus en popularité comme le montre les travaux les plus récents de Ribeiro *et al.* [131].

2.2.3.4 Recherche avec Tabous

Parmi les autres métaheuristiques dites "classiques", la recherche avec tabous (RT) a été proposée en 1986 par Fred Glover [65]. Le principe de cette méthode est le suivant : à solution donnée, on explore les solutions qui lui sont proches, c'est-à-dire celles qu'on peut atteindre par de simples modifications de la solution initiale. Cet ensemble de solutions est appelé voisinage. Une particularité est toutefois de garder en mémoire, sous forme d'une liste

taboue, les espaces déjà explorés pour éviter d'y retourner.

L'utilisation de cette métaheuristique a été un tournant important dans la résolution de problèmes dans les sciences appliquées, la gestion et l'ingénierie. La recherche avec tabous possède des liens avec les algorithmes dits "évolutionnaires" (dont les algorithmes génétiques) [67]. La mémoire adaptative de l'approche taboue a aussi attiré les chercheurs puisqu'elle permet dans la pratique une amélioration des réseaux de neurones [124]. Les applications actuelles de la recherche avec tabous sont dans la planification de ressources, les télécommunications, l'analyse financière, l'ordonnancement, la planification d'espace, la distribution d'énergie, l'ingénierie moléculaire etc. Plusieurs publications et études ont porté sur la recherche avec tabous et ont contribué à étendre la version de base et permettre de trouver des solutions dont la qualité est supérieure à celle obtenue auparavant [64]. La recherche avec tabous se distingue des autres métaheuristiques par son exploitation de la mémoire. Cependant, cela a aussi contribué à la découverte de stratégies potentielles de gestion de mémoire. Les nouvelles données et principes qui ont émergé de la recherche avec tabous ont donné la base de création à des systèmes dont les capacités sont supérieures à leurs antécédents en terme de temps de réponse, de qualité de solution etc. De plus, il reste plusieurs variations qui n'ont pas encore été explorées qui pourraient faire avancer le domaine encore plus [66]. Par exemple, la concentration sur les bonnes régions, l'identification des régions prometteuses, les modèles de recherches non monotones et l'intégration et l'extension des solutions. La Figure 2.34 illustre un algorithme de recherche avec tabous. L'algorithme commence par générer aléatoirement une solution initiale s et la désigne comme meilleure solution connue s^*. Une initialisation se fait au niveau de la liste taboue et du compteur k. L'algorithme détermine ensuite la meilleure solution (s_{k+1}) dans le voisinage N de s. Si la solution trouvée est meilleure que s^* alors s_{k+1} remplace s^* et la liste taboue est mise à jour. L'algorithme réitère ce processus jusqu'à ce que la condition d'arrêt soit satisfaite.

Crainic *et al.* [34] ont proposé une taxonomie de parallélisation de la recherche avec tabous. Ce travail est considéré comme le travail le plus complet sur le domaine et se base sur une classification à trois dimensions. La première, appelée contrôle de cardinalité, définit si la recherche est contrôlée par un seul processus ou encore par plusieurs qui peuvent collaborer

```
Générer une solution initiale s
s*=s
k=0
Initialiser la liste taboue
Tant que (Condition d'arrêt non satisfaite)
        Déterminer la meilleure solution (s_{k+1}) dans N(s_k)
            en tenant compte de la liste taboue
        Si f(s_{k+1})<f(s*)
                s*=s_{k+1}
        k=k+1
        Mise à jour de la liste taboue
```

Figure 2.34 – Algorithme de recherche avec tabous [143]

ou non. Dans le deuxième cas, chaque processus est responsable de sa propre recherche et d'établir les communications avec les autres processus. La recherche globale se termine en même temps que les recherches individuelles. La deuxième dimension se réfère au type et à la flexibilité de la recherche. Elle est traitée en quatre niveaux qui correspondent chacun à des schémas de contrôles. La troisième dimension est la différenciation de la recherche. Une revue de littérature complète peut être retrouvée dans Gendreau [64]. L'auteur décrit notamment les versions hybrides de recherche avec tabou comme étant les plus prometteuses du domaine. C'est un constat qui s'est confirmé dans les récents travaux comme ceux de Mack *et al.* [99] ou encore Homberger *et al.* [85].

2.2.4 Autres métaheuristiques parallèles

Les métaheuristiques vues préalablement, mêmes si elles sont les plus populaires, ne forment pas une liste exhaustive des métaheuristiques qui existent. D'autres travaux orientés vers la parallélisation des métaheuristiques ont été conduits.

On peut citer les algorithmes d'estimation de distribution (Estimation of Distribution Algorithms EDA) [112] et [111] qui font partie de la famille des algorithmes évolutionnaires. Ces algorithmes agissent sur une population d'individus et estiment la probabilité de distribution de chaque variable dans ces individus. Cette estimation est par la suite utilisée pour générer un nouvel ensemble d'individus qui tend à être plus proche de l'optimum du problème. Cet algorithme a la particularité d'être exigeant en coûts de calcul, principalement pour la fonction d'estimation. C'est justement cette particularité qui a été le point de départ de l'approche

parallèle de ces algorithmes. Les recherches dans le domaine des algorithmes d'évaluation de distribution sont plutôt récentes. On cite notamment les travaux d'Ahm *et al.* [4] qui ont introduit un framework pour développer des algorithmes d'estimations parallèles et ceux de Mendiburu *et al.* [105] qui ont étendu des algorithmes déjà existants en utilisant de la programmation distribuée.

On cite aussi comme métaheuristique parallèle, la recherche par dispersion parallèle (Parallel Scatter search) qui fait également partie de la famille des algorithmes évolutionnaires. La version séquentielle de cet algorithme, introduite par Laguna *et al.* [91], fait combiner par un processus intelligent, un nombre de solutions jusqu'à atteindre un optimum. La principale différence avec les algorithmes génétiques, est que la recherche de l'optimum local est guidée, dans le sens qu'un échantillon référence (RefSet) est sélectionné parmi l'ensemble de la population. Cet échantillon est intensifié et mis à jour à chaque itération. Une fois que les solutions de cet échantillon sont combinées, une recherche locale est lancée pour tenter d'améliorer les résultats obtenus. L'échantillon est mis à jour avec autant les bonnes que les mauvaises solutions. La version parallèle de la recherche par dispersion est une évolution naturelle de l'algorithme comme moyen efficace de contrer le temps de calcul. Le parallélisme peut être introduit de plusieurs manières, soit en lançant plusieurs algorithmes de recherche locale sur plusieurs processeurs, soit en divisant une même procédure de recherche locale sur plusieurs processeurs. La principale application de cet algorithme a été pour la résolution du problème de la médiane p (The p-median problem) notamment grâce aux travaux de Garcia *et al.* [61].

Une métaheuristique plus récente est celle de la recherche à voisinage variable parallèle (Parallel Variable Neighborhood Search) de Hansen *et al.* [79] qui repose sur le principe de changement systématique de voisinage tant dans la descente vers les optimums locaux, que dans la sortie des vallées qui les contiennent. La parallélisation de cet algorithme sert soit à augmenter la taille du problème à résoudre, soit à limiter le temps de calcul de la recherche locale ou encore à étendre l'espace de recherche. Plusieurs versions parallèles de la recherche à voisinage variable ont vu le jour [110]. La première, appelée version synchrone, permet de

réduire le temps d'exécution de la recherche locale. Il s'agit simplement de distribuer la partie recherche locale entre les processeurs. La deuxième, appelée version répliquée, consiste à lancer en parallèle plusieurs instances de l'algorithme séquentiel, et ceci dans le but d'explorer le plus d'espace de recherche possible.

Pour mieux décrire l'ensemble des métaheuristiques parallèles, on a recours à des classifications. De plus, le nombre croissant de versions dérivées implique la nécessité de regroupement en classes. C'est aussi un moyen efficace de distinguer les métaheuristiques standards et les hybrides. Dans la section suivante, deux classifications sont présentées.

2.2.5 Classification des métaheuristiques parallèles

Les recherches les plus élaborées en ce qui concerne la classification des métaheuristiques parallèles sont celles effectuées par Crainic et Toulouse [33] et Cung *et al.* [36]. Ces deux classifications n'abordent pas la question des métaheuristiques parallèles de la même manière. C'est pour cela que ces deux classifications sont présentées séparément.

2.2.5.1 Classification de Crainic et Toulouse

La classification de Crainic et Toulouse considère les sources de parallélisation dans les métaheuristiques. Selon le niveau dans lequel le traitement parallèle se produit, trois types de parallélisation sont identifiés.

Le premier type (aussi appelé parallélisme de bas niveau) est une stratégie de parallélisation au sein d'une itération de la métaheuristique. Elle a pour but de réduire le temps d'exécution de la version séquentielle de la même métaheuristique. Elle ne cherche ni à améliorer la qualité des solutions, ni à mieux explorer l'espace de recherche.

Le deuxième type est un parallélisme qui concerne les variables de décisions. Cette stratégie de parallélisation est retrouvée dans la littérature sous la forme de métaheuristiques en mode maître-esclave. Le noeud maître distribue les variables parmi les autres noeuds disponibles et ceux-ci procèdent à l'exploration de manière concurrente et indépendante. Le noeud maître récupère ensuite les données partielles des esclaves et les recombine.

Enfin, le troisième type de parallélisation est un stratégie d'exploration multiple à

l'aide de plusieurs processus concurrents. Il est possible que les processus n'utilisent pas la même heuristique. Si les processus communiquent entre eux, la stratégie est appelée coopérative. Par contre, si aucune information n'est échangée entre les processus, la stratégie est appelée indépendante.

2.2.5.2 Classification de Cung et al.

La classification de Cung *et al.* se veut indépendante de l'architecture. Elle ramène les métaheuristiques basées sur la recherche locale à un graphe. Dans ce graphe, les noeuds sont des solutions et les arcs relient les solutions voisines. Les métaheuristiques étant itératives, les itérations sont traduites par des évaluations successives du voisinage de la solution. Ces évaluations sont suivies d'un déplacement vers une direction ou une autre du graphe en évitant les minima locaux. L'ensemble des solutions visitées tout au long de ce processus est appelé marche. Selon comment la marche est conduite, deux grandes classes sont identifiées : marche simple et marche multiple.

La marche simple concerne les métaheuristiques qui explorent le voisinage des solutions avec une trajectoire unique. Leur parallélisation se fait sur la recherche du meilleur voisin, sur la fonction objectif ou encore sur la décomposition du domaine.

La marche multiple concerne les métaheuristiques qui explorent plusieurs trajectoires en leur attribuant chacun un processus. Chaque processus qui entame une marche est appelé thread. Si les threads s'échangent leurs informations respectives, alors une sous-classe appelée marche multiple coopérative est introduite. Si les threads sont isolés alors c'est une marche multiple indépendante.

En plus de disposer de plusieurs classifications, les métaheuristiques parallèles font l'objet d'analyses de résultats poussées. Par opposition aux méthodes exactes où seul importe le temps d'exécution, les métaheuristiques parallèles doivent à la fois satisfaire les critères de rapidité et d'éloignement par rapport à l'optimum. C'est pour cela que les métaheuristiques parallèles disposent également de mesures de performances qui renseignent sur les gains par rapport à leurs homologues séquentielles. Un bref survol de ces mesures est fait dans la section suivante.

2.2.6 Les mesures de performance des métaheuristiques parallèles

Les mesures de performances vues à la Section 2.1.5 sont aussi valables pour mesurer celles des métaheuristiques parallèles. Cependant, comme les métaheuristiques ne sont pas des algorithmes exacts, il convient de nuancer certaines mesures, voir même de les réajuster pour qu'elles conviennent mieux à comparer les métaheuristiques entre elles. Alba *et al.* [9] précisent que pour les algorithmes non déterministes, comme les métaheuristiques, c'est le temps moyen des deux versions séquentielles et parallèles qui doit être pris en compte. Il propose ainsi différentes définitions de l'accélération. Une accélération forte qui compare l'algorithme parallèle au résultat du meilleur algorithme séquentiel connu. C'est ce qui correspond le plus à la vraie définition de l'accélération mais vu la difficulté de trouver à chaque fois le meilleur algorithme existant, cette norme n'est pas beaucoup utilisée. Une accélération dite faible compare l'algorithme parallèle avec la version séquentielle développée par le même chercheur. Il peut alors présenter ses progrès soit en terme de qualité, soit en accélération pure. Barr et Hickman [19] ont présenté une taxonomie différente qui consiste en accélération, accélération relative et accélération absolue. L'accélération relative traite du rapport entre la version parallèle exécutée sur un seul processeur et celle exécutée sur l'ensemble des processeurs. Enfin, l'accélération absolue, qui est le rapport de la version séquentielle la plus rapide sur n'importe quelle machine et le temps d'exécution de la version parallèle.

2.2.7 Conclusion

Presque toutes les métaheuristiques classiques ont donné naissance à des versions parallèles plus ou moins fidèles à leur origine. Ce constat traduit une nécessité de vitesse et de performance que les ordinateurs parallèles tentent de fournir à des problèmes de plus en plus complexes et à des exigences de temps de plus en plus restrictives. Aucune métaheuristique ne peut être qualifiée de "meilleure" par rapport à une autre, mais la préférence va selon la nature du problème à résoudre.

2.3 Les objectifs de la recherche

L'objectif principal de ce mémoire est de concevoir une métaheuristique parallèle pour solutionner un problème d'optimisation combinatoire. Le problème choisi est un problème d'ordonnancement sur machine unique avec temps de réglages dépendants de la séquence. Ce problème sera détaillé à la Section 3.1.

De manière générale, un problème d'ordonnancement consiste à planifier l'exécution d'un certain nombre de tâches en cherchant à optimiser un ou plusieurs objectifs particuliers. Le problème d'ordonnancement traité dans ce mémoire appartient à la classe des problèmes NP-Difficiles. De ce fait, la recherche de solutions au moyen de métaheuristiques est tout à fait justifiée et nous avons opté pour les algorithmes génétiques pour leur grande adaptabilité et leur côté évolutionnaire. Compte tenu du besoin de calculs importants, nous visons à concevoir une métaheuristique parallèle exploitant certaines des structures et des modèles de parallélisation présentés à la Section 2.1.

Les objectifs spécifiques de ce mémoire sont les suivants :

1. Mettre au point un algorithme génétique séquentiel pour la résolution du problème d'ordonnancement et de comparer son efficacité sur des problèmes tests de la littérature.

2. Concevoir une version parallèle de cette métaheuristique dans un modèle à multiples populations et avec une topologie en anneau. Ainsi, il sera possible de mesurer l'impact de la version parallèle tant sur la qualité de solutions que sur le temps d'exécution.

Chapitre 3

Conception d'un algorithme génétique parallèle pour le problème d'ordonnancement sur machine unique avec temps de réglage dépendant de la séquence

3.1 Description du problème d'ordonnancement sur machine unique avec temps de réglage dépendant de la séquence

Dans certains contextes industriels, les unités de production sont bâties autour d'une seule ressource avec des traitements successifs. On peut citer, par exemple, des domaines comme la métallurgie, la coloration du plastique ou encore la fabrication de papier. Ce cas de figure implique qu'une seule commande (job) peut être traitée à la fois et, qu'une fois lancée, la commande ne peut être interrompue. Une autre particularité réside dans les temps de réglages entre deux commandes pendant lesquels la ressource est inutilisable.

L'utilisation de la ressource est régie par un carnet de commandes. Chaque commande est identifiée par un numéro (de 1 à N), un temps de traitement, une date de livraison, un temps de réglage initial et des temps de réglage relatifs aux autres commandes. L'ordonnancement vise à déterminer la séquence de production permettant d'optimiser un objectif particulier. Dans ce mémoire, nous nous intéressons à la minimisation du retard total qui s'exprime par la somme des retards de chaque commande de la manière suivante :

$$T_Q = \sum_{k=1}^{N} t_{Q(j)} \tag{3.1}$$

où $t_{Q(j)}$ est le maximum entre 0 et la différence entre la date de fin de traitement ($c_{Q(j)}$) et la date de livraison ($d_{Q(j)}$) de la commande j selon l'équation suivante :

$$t_{Q(j)} = max\{0, c_{Q(j)} - d_{Q(j)}\} \tag{3.2}$$

Parmi les principaux travaux ayant traités de cette problématique, on cite ceux de Rubin et Ragatz [134], Gagné *et al.* [59] [60], Koulamas [90], Ragatz [128], Tan *et al.* [146], Lee *et al.* [93] [147], França *et al.* [58], Luo *et al.* [97] [96] ainsi que Gupta *et al.* [78]. Une classification de tous les problèmes d'ordonnancement ayant ou non une dépendance avec la séquence a été proposée par Allahverdi *et al.* [11]. Dans la prochaine section, nous nous intéressons plus particulièrement aux travaux de Rubin et Ragatz [134] qui ont utilisé un algorithme génétique pour solutionner ce problème.

3.2 L'algorithme génétique de Rubin et Ragatz

Rubin et Ragatz [134] ont abordé le problème de l'ordonnancement sur machine unique avec temps de réglage dépendant de la séquence avec un algorithme génétique classique tel que décrit à la Figure 2.15. La fonction objectif adoptée est la minimiation du retard total. Les caractéristiques de cet algorithme sont décrites dans les paragraphes qui suivent.

Premièrement, la taille de la population a été fixée à 40. Selon les auteurs, une population de 20 donnait de mauvaises performances alors qu'une population de 60 ou encore 80 n'amenait aucune amélioration par rapport au taux de convergence. De plus, comme le temps de calcul augmente de façon linéaire avec la taille de population, il n'est pas bénéfique de trop augmenter la taille de la population.

La méthode de sélection adoptée par les auteurs est celle par roulette. La probabilité de sélection d'un individu est inversement proportionnelle à sa fonction objectif. Une fonction de transformation a aussi été implémentée pour éviter les divisions par 0.

L'opérateur de croisement implémenté par Rubin et Ragatz est particulier et ne correspond à aucun de ceux décrits à la Section 2.2.2.1. Aucune explication n'a été fournie par les auteurs sur le choix de cet opérateur par rapport à un autre de la littérature. Toutefois, ils

ont mentionné avoir préféré cette codification par rapport à la codification binaire pour éviter d'avoir des vecteurs trop volumineux. Un exemple de cet opérateur de croisement est présenté à la Figure 3.1. Tout d'abord, deux groupes de commandes sont formés. À la Figure 3.1 partie (a), le groupe g1 est celui des commandes impaires et le groupe g2 est celui des commandes paires. Le choix des groupes est aléatoire mais il est préférable que les deux groupes soient de même taille. L'ordre selon lequel chaque groupe apparaît dans chacun des parents est noté dans la partie (b). Toutes les combinaisons sont rangées en colonnes pour une meilleure visibilité. Ensuite, les positions auxquelles apparaissent les éléments des deux groupes sont notées dans la partie (c) du schéma. Ces positions sont représentées par des lettres pour éviter la confusion avec les numéros des commandes. Enfin, pour constituer les enfants, on effectue la combinaison des différents ordres et positions. Par exemple, l'enfant E1 est constitué par les éléments de g1 dans l'ordre qu'ils apparaissent dans P1 aux positions issues de P1. Il est aussi constitué du groupe g2 dans l'ordre de P1 aux positions de P1. Par conséquent, l'enfant E1 va être un clone de P1. Les différentes combinaisons donnent au total huit enfants dont deux clones des parents.

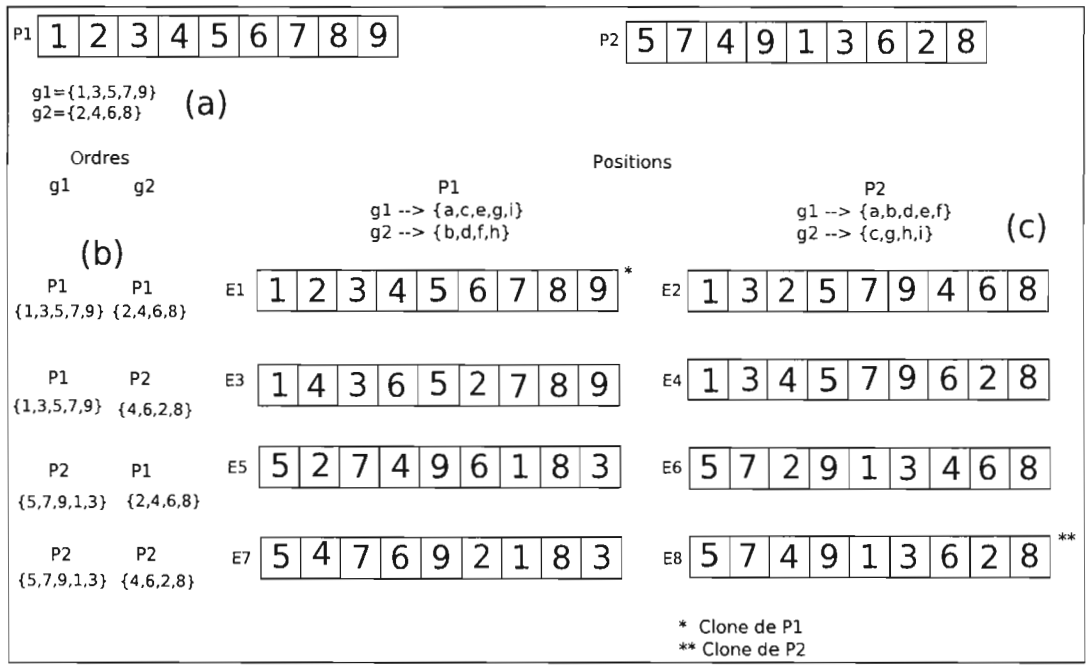

Figure 3.1 – Opérateur de croisement Rubin & Ragatz [134]

La mutation a aussi été abordée par Rubin et Ragatz de manière particulière. L'opérateur de mutation ne correspond à aucun de ceux vus à la Section 2.2.2.1. En effet, c'est une recherche locale qui consiste à permuter toutes les positions adjacentes et conserver une permutation seulement si elle améliore la fonction fitness de l'individu. Ce genre de mutation est beaucoup plus lent à l'exécution que la version traditionnelle mais, selon les auteurs, assure une amélioration de la majorité des individus mutés. Le taux de mutation a été fixé à 10% à chaque génération.

Enfin, la phase de remplacement a été abordée de manière élitiste et la gestion de la diversité s'est faite par un procédé nommé *immigration* et qui consiste à la régénération aléatoire des 10% pires individus à chaque génération.

Nous avons, dans un premier temps, tenté de reproduire l'algorithme génétique proposé par ces auteurs. Cet algorithme, noté AG0 dans la suite du document. a été évalué sur un ensemble de problèmes tests proposés par Ragatz [127] pour valider son fonctionnement. Ces problèmes ont spécialement été générés pour l'ordonnancement sur machine unique avec temps de réglage dépendant de la séquence avec l'objectif de minimiser le retard total. Les problèmes tests se composent de quatre groupes selon le nombre de commandes qu'ils contiennent. Ces groupes sont respectivement de 15, 25, 35 et 45 commandes. Pour chaque taille de problème, huit problèmes sont générés en faisant varier trois paramètres. Le premier paramètre, "Processing Time Variance" (PTV), définit la variabilité du temps d'exécution et peut prendre deux valeurs "low" et "high". Le deuxième paramètre, "Tardiness Factor" (TF), correspond au degré de retard dans les problèmes générés et peut prendre deux valeurs "low" et "medium". Enfin, le troisième paramètre, "Due Date Range" (DDR), correspond à la variabilité dans les dates d'échéance et peut être "narrow" ou "wide".

Rubin et Ragatz ont fixé le nombre total de générations à 300 ou jusqu'à ce que les résultats du Branch & Bound soient atteints ou améliorés. En effet, les auteurs ont conçu une procédure de type Branch & Bound limitée à l'exploration de deux millions de noeuds. Ces solutions servent de base de comparaison pour la performance de l'algorithme génétique.

Le Tableau 3.1 présente les écarts absolus avec la soluion de référence issus de l'article

de Rubin et Ragatz [134] et ceux obtenus par l'AG0 pour 300 et 1000 générations. Tout comme Rubin et Ragatz, les résultats de l'AG0 représentent une moyenne de 20 essais. En terme de qualité, trois mesures sont prises en compte : le meilleur résultat obtenu (Meilleur), le résultat médian (Médian) et le pire résultat obtenu (Pire). Pour la comparaison de performance entre l'AG0 (300 générations) et l'algorithme génétique de Rubin & Ragatz, on observe, en utilisant un écart maximal de 5% entre les résultats médians, une performance identique sur 25 des 32 problèmes. Pour les 7 autres problèmes dont les résultats sont présentés en caractères gras, l'AG0 est moins performant, ce qui laisse croire que certains éléments diffèrent entre les deux algorithmes. L'augmentation du nombre de générations à 1000 corrige cette situation et fait même en sorte que les résultats sont améliorés pour 5 problèmes. Ces problèmes sont indiqués en caractères gras dans le Tableau 3.1. On trouve, en moyenne, la meilleure solution, après 830 générations, ce qui explique le nombre maximal de générations fixé à 1000.

À la section suivante, nous décrivons le fonctionnement d'un autre algorithme génétique ayant l'algorithme AG0 comme point de départ, mais utilisant des opérateurs de croisement classiques ainsi que des paramètres différents. Une analyse comparative de la performance sera réalisée avec l'AG0.

3.3 Conception d'un algorithme génétique séquentiel

L'algorithme génétique présenté dans cette section utilise des opérateurs génétiques classiques et a été conçu en suivant des indications retrouvées dans la littérature pour la fixation des différents paramètres. Cet algorithme est noté AG1.

La taille de la population est fixée égale au nombre de commandes du problème et le nombre de générations est fixé à 1000 selon l'étude de convergence réalisée. Des essais numériques, faisant varier le taux de mutation entre 10 et 20 %, ont également permis de fixer le taux de mutation à 19%. Cette mutation est identique à celle de l'AG0 et utilise le principe de la recherche locale introduit par Rubin et Ragatz [134].

L'opérateur de croisement a été défini après l'expérimentation des opérateurs suivants : MPX, OX, LOX et CX. Comme aucun de ces opérateurs ne s'est particulièrement distingué,

Prb	# comm.	B&B	*	GA (Rubin & Ragatz) Meilleur	Médian	Pire	AG0(300) Meilleur	Médian	Pire	AG0(1000) Meilleur	Médian	Pire
PROB401	15	90	*	0,0	4,4	4,4	0,0	6,7	27,8	4,4	4,4	21,1
PROB402	15	0	*	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
PROB403	15	3418	*	0,0	0,0	0,0	0,0	1,1	2,3	0,0	2,1	3,7
PROB404	15	1067	*	0,0	0,0	0,0	0,0	0,0	6,8	0,0	0,0	0,0
PROB405	15	0	*	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
PROB406	15	0	*	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
PROB407	15	1861	*	0,0	0,0	0,0	0,0	0,7	2,1	0,0	0,7	2,7
PROB408	15	5660	*	0,0	0,0	0,9	0,0	0,0	1,5	0,0	1,3	2,5
PROB501	25	264		0,0	1,5	3,8	1,1	7,2	12,1	2,3	4,2	8,3
PROB502	25	0	*	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
PROB503	25	3511		-0,4	0,2	0,9	-0,1	1,9	3,5	-0,1	0,4	4,9
PROB504	25	0	*	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
PROB505	25	0	*	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
PROB506	25	0	*	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
PROB507	25	7225	*	2,1	6,1	9,6	0,4	4,9	9,3	1,2	3,0	5,3
PROB508	25	2067		-5,9	-5,9	-1,5	-7,4	-1,9	15,6	-7,4	-1,7	5,2
PROB601	35	30		76,7	150,0	193,3	73,3	**243,3**	343,3	53,3	**140,0**	236,7
PROB602	35	0	*	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
PROB603	35	17774		-0,7	0,4	2,2	1,8	**5,8**	7,9	0,6	2,9	5,6
PROB604	35	19277		0,2	1,0	2,6	1,4	5,8	11,4	0,3	2,7	4,5
PROB605	35	291		13,7	37,3	56,7	11,7	**54,0**	97,3	5,5	**25,8**	45,4
PROB606	35	0	*	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
PROB607	35	13274		5,0	6,6	7,6	0,9	6,4	8,8	1,1	2,5	3,7
PROB608	35	6704		-29,0	-28,6	-26,7	-28.2	**-11,6**	-4,0	-28,0	**-23,3**	-14,4
PROB701	45	116		57,8	82,8	118,1	85,3	**131,9**	175,9	59,5	80,2	136,2
PROB702	45	0	*	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
PROB703	45	27097		-0,1	1,5	2,5	3,3	5,9	9,1	1,0	3,2	4,6
PROB704	45	15941		-2,4	-1,6	1,0	3,9	**12,1**	22,3	-0,8	3,8	9,1
PROB705	45	234		53,4	89,3	114,5	69,7	**95,7**	145,7	48,3	**69,2**	91,5
PROB706	45	0	*	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
PROB707	45	25070		-1,1	1,8	6,3	0,3	5,4	8,8	-2,3	1,3	5,0
PROB708	45	24123		2,8	7,2	10,1	-1,4	6,6	11,8	-2,2	**0,5**	3.4

* Solution Optimale

Tableau 3.1 – Résultats AG0

l'AG1 utilise aléatoirement, à chaque génération, l'un de ces 4 opérateurs pour garantir un maximum de diversité dans le processus de reproduction. Cette combinaison d'opérateurs a produit les meilleurs résultats.

Pour les essais numériques avec l'AG1, quatre autres groupes de problèmes de tailles respectives 55, 65, 75 et 85 commandes ont été ajoutés. Ces problèmes tests ont été proposés par Gagné *et al.* [59] et vont permettre d'analyser le comportement de l'algorithme sur des problèmes de plus grande taille et ainsi réaliser une meilleure comparaison entre l'AG0 et

l'AG1. Le Tableau 3.2 présente les résultats obtenus sur les 64 problèmes.

On observe, en utilisant un écart maximal de 5% entre les résultats médians, une meilleure performance de l'AG0 par rapport à l'AG1 sur 4 problèmes pour les groupes de taille 15 à 45 commandes. On constate toutefois que AG1 s'avère plus performant sur 6 problèmes lorsque l'on considère les groupes de taille 55 à 85 commandes tout en donnant un résultat identique pour les 26 autres problèmes. On note plus particulièrement que les gains de performance sont très importants sur le problème PROB655. Il y a également eu des améliorations sur les meilleurs résultats obtenus pour 4 problèmes (PROB651, PROB655, PROB658 et PROB751) et une amélioration des pires résultats pour 7 problèmes (PROB551, PROB651, PROB655, PROB658, PROB751. PROB754 et PROB855).

Si on considère l'ensemble des 64 problèmes, on constate, sur la base des résultats médians, que l'AG1 donne les mêmes performances que l'AG0 sur 54 d'entre eux, améliore les résultats médians de 6 problèmes (en gras dans la colonne AG1). Seulement 4 problèmes obtiennent de meilleurs résultats avec l'AG0 (en gras dans la colonne AG0). Les résultats de la colonne "Meilleur" et "Pire" sont également intéressants pour l'AG1. Dans la colonne "Meilleur", 6 problèmes ont été améliorés, 56 ont été atteints et 2 sont pires que l'AG0. Enfin, la colonne "Pire" présente, pour l'AG1, 9 problèmes améliorés, 51 atteints et 4 pires que l'AG0. L'algorithme AG1 est ainsi retenu dans la conception de l'approche de parallélisation décrite à la section suivante.

3.4 Parallélisation de l'algorithme génétique

L'un des objectifs spécifiques de ce mémoire est d'introduire le parallélisme dans les algorithmes génétiques. Nous utilisons un modèle en îlots avec une topologie en anneau unidirectionnel comme forme de parallélisation de l'algorithme génétique AG1. C'est un modèle qui a fait ses preuves dans la littérature et présente un moyen intéressant d'améliorer les performances. Plusieurs travaux tels ceux de Grosso [76], Pettey *et al.* [119], Tanese [148], Beldin [20] et Anderson *et al.* [14] ont adopté ce modèle.

Le fonctionnement de l'algorithme parallèle est le suivant : l'algorithme AG1 est

Prb	# comm.	B&B	*	AG0(1000) Meilleur	Médian	Pire	AG1 Meilleur	Médian	Pire
PROB401	15	90	*	4,4	**4,4**	21,1	4,4	11,1	37,8
PROB402	15	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB403	15	3418	*	0,0	2,1	3,7	0,0	1,9	3,4
PROB404	15	1067	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB405	15	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB406	15	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB407	15	1861	*	0,0	0,7	2,7	0,0	1,2	2,9
PROB408	15	5660	*	0,0	1,3	2,5	0,0	1,5	2,5
PROB501	25	264		2,3	4,2	8,3	3,0	4,9	12,1
PROB502	25	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB503	25	3511		-0,1	0,4	4,9	-0,1	0,6	2,2
PROB504	25	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB505	25	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB506	25	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB507	25	7225	*	1,2	3,0	5,3	0,5	2,8	5,2
PROB508	25	2067		-7,4	**-1,7**	5,2	0,0	5,3	11,7
PROB601	35	30		53,3	140,0	236,7	**43,3**	140,0	236,7
PROB602	35	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB603	35	17774		0,6	2,9	5,6	1,4	3,2	4,6
PROB604	35	19277		0,3	2,7	4,5	1,3	2,7	4,7
PROB605	35	291		5,5	25,8	45,4	6,9	23,7	**39,2**
PROB606	35	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB607	35	13274		1,1	2,5	3,7	-0,3	3,1	9,2
PROB608	35	6704		-28,0	-23,3	-14,4	-28,4	-23,0	-16,9
PROB701	45	116		59,5	**80,2**	136,2	**35,3**	85,3	**119,0**
PROB702	45	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB703	45	27097		1,0	3,2	4,6	1,4	2,8	4,8
PROB704	45	15941		-0,8	3,8	9,1	0,7	4,8	10,0
PROB705	45	234		48,3	**69,2**	91,5	45,3	77,8	106,0
PROB706	45	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB707	45	25070		-2,3	1,3	5,0	-1,2	0,5	3,1
PROB708	45	24123		-2,2	0,5	3,4	-3,4	0,2	5,5
PROB551	55	212		97,3	154,6	288,6	95,8	**128,8**	**162,3**
PROB552	55	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB553	55	40828		4,7	5,9	7,7	3,7	5,3	6,6
PROB554	55	15091		11,0	17,3	25,0	8,3	15,0	20,0
PROB555	55	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB556	55	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB557	55	36489		4,3	7,2	10,1	2,5	4,9	8,4
PROB558	55	20624		6,7	12,8	18,4	2,6	8,1	13,7
PROB651	65	295		96,6	136,3	178,2	**72,2**	**112,2**	**168,5**
PROB652	65	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB653	65	57779		4,8	6,8	8,9	5,3	6,3	10,0
PROB654	65	34468		8,9	14,6	18,8	7,6	11,3	15,7
PROB655	65	13		5300,0	7800,0	9500,0	**853,8**	**1107,7**	**1961,5**
PROB656	65	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB657	65	56246		5,7	7,5	10,6	4,0	5,4	6,8
PROB658	65	29308		9,1	15,1	21,6	**1,9**	**6,8**	**11,9**
PROB751	75	263		162,7	198,3	244,0	126,2	171,1	197,0
PROB752	75	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB753	75	78211		5,7	7,8	9,1	5,0	6,4	7,8
PROB754	75	35826		11,9	17,8	26,1	12,4	15,7	**19,8**
PROB755	75	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB756	75	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB757	75	61513		6,6	8,9	11,8	3,7	5,3	7,7
PROB758	75	40277		9,1	19,0	21,8	7,1	**11,4**	17,0
PROB851	85	453		126,9	165,6	194,3	123,4	164,2	191,2
PROB852	85	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB853	85	98540		5,0	6,8	8,2	4,9	7,0	9,0
PROB854	85	80693		7,7	9,8	12,4	7,4	9,3	12,1
PROB855	85	333		118,3	157,1	200,9	137,5	158,6	**191,3**
PROB856	85	0	*	0,0	0,0	0,0	0,0	0,0	0,0
PROB857	85	89654		4,2	6,1	9,3	4,5	6,4	8,9
PROB858	85	77919		5,4	8,9	11,5	5,1	8,2	11,1

* Solution Optimale

Tableau 3.2 – Résultats AG1

dupliqué sur un nombre de processeurs défini et chaque processeur dispose de sa population propre ou sous-population. Les sous-populations sont reliées entre elles en forme d'anneau. À un intervalle donné I, chaque processeur envoie une copie de ses R meilleurs individus à son voisin de droite, et va, par conséquent, recevoir R individus qu'il intègre dans sa sous-population en remplaçant ses pires individus. Lorsque le nombre total de générations est effectué, chaque sous-population est triée et les meilleurs individus issus de ces sous-populations sont eux-mêmes rassemblés et triés. Cette version parallèle est notée AGP0.

L'exécution de l'algorithme AGP0 s'est faite sur une machine parallèle possédant les caractéristiques suivantes : un noeud principal AMD Opteron(tm) Processeur 250 (64 bits) 2.4GHz avec 4Go de mémoire vive sur Fedora Core 3, deux noeuds quadruple processeurs Quad AMD Opteron(tm) Processeur 880 (64 bits) 2.4GHz avec 16Go de mémoire vive sur CentOS 4.3 et soixante six noeuds Dual AMD Opteron(tm) Processeur 250 (64 bits) 2.4GHz avec 4Go de mémoire vive. C'est le même environnement d'exécution utilisé pour l'AG0 et l'AG1 à l'exception du nombre de processeurs sélectionné via le mode de soumission. Cependant, pour garder les vitesses de communications homogènes, l'exécution s'est déroulée seulement sur les noeuds biprocesseurs en spécifiant l'utilisation d'un seul processeur par noeud.

L'implémentation de l'AGP0 s'est faite dans le langage C et à l'aide du standard MPI décrit à la Section 2.1.4.2. De plus, comme l'environnement matériel permet l'utilisation du réseau Infiniband [86], on a pu utiliser la dernière version mvapich 2 [114] qui est l'implémentation de MPI pour ce type de réseau. De même, la compilation s'est faite avec le compilateur mvapich2-0.9.8. Diverses fonctions MPI ont été utilisées pour gérer les communications entre les processeurs et sont citées durant les expérimentations.

L'exécution sur plusieurs processeurs amène la possibilité de parcourir l'espace de recherche de plusieurs manières différentes. Dans ce qui suit, deux manières vont être abordées : la première consiste à subdiviser la taille de la population sur le nombre de processeurs (stratégie POP), tandis que la deuxième réduit le nombre de générations (stratégie GEN). Les deux stratégies ont leurs avantages et leurs inconvénients comme on pourra le constater ultérieurement.

Dans les essais numériques réalisés, les problèmes dont les solutions sont déjà égales

à 0 par la version AG1 ne sont pas considérés puisque aucune amélioration ne peut y être apportée par la version parallèle. L'ensemble des problèmes tests est ainsi réduit à 43 problèmes. Le plan d'expérience fait varier le nombre de processeurs (1, 2 et 4) et ajuste, selon la stratégie, la taille de la sous-population sur chaque processeur ou le nombre de générations. La raison pour laquelle on ne dépasse pas 4 processeurs est la faible taille des problèmes. Le plan d'expérience comprend également la division des exécutions en deux grandes catégories : celles sans échanges d'information entre les processeurs et celles avec échanges. Pour les tests avec échanges, différentes fréquences (à toutes les 1, 5, 10, 20, 50,100 et 500 générations) et différents taux d'échanges (5%, 10% et 50% de la population échangée) sont évalués.

3.4.1 AG sans échange d'information

L'implémentation de l'AGP0 sans échange est un moyen de connaître le coût des communications. Cet ensemble de tests sert de plancher pour la qualité des solutions car on suppose, a priori, que les échanges sont bénéfiques pour la qualité. Il sert également de plafond pour les accélérations car, plus il y a d'échanges, plus il y a de ralentissements.

Pour la comparaison de la qualité des solutions, nous utilisons les mêmes critères que pour la comparaison de l'AG1 avec l'AG0, à savoir le résultat "Meilleur", "Médian" et "Pire". En premier lieu, on constate au Tableau 3.3 que les résultats de l'AGP0 utilisant 2 et 4 processeurs, selon la stratégie de subdivision de la population (POP) et de maintien du nombre de générations, ne permettent pas de conserver la qualité obtenue par l'AG1. En effet, un seul "Meilleur" et un seul "Pire" ont été améliorés (PROB655) alors qu'aucun résultat médian n'a été amélioré. Dans la majorité des cas, on observe une dégradation de la qualité des résultats par rapport à l'AG1. C'est la conséquence de la subdivision de la population et de l'isolement des sous-populations par l'absence d'échanges d'information.

En comparant les résultats de l'AGP0 exécuté sur 2 et 4 processeurs, on remarque que les résultats obtenus sont légèrement de moins bonne qualité sur 4 processeurs. Ainsi, on observe que le nombre de résultats médians atteints chute de 23 à 14 en considérant un écart maximal de 5% entre les résultats médians. Ce résultat peut être interprété de la manière

suivante : lorsqu'une population est isolée, elle a plus de chance de se diversifier si elle dispose d'un grand nombre d'individus. Cette diversité permet aux opérateurs de croisement d'être plus efficaces. Dans ce cas de figure, deux sous-populations de $N/2$ individus donnent de meilleurs résultats que quatre sous-populations de $N/4$ individus. C'est la raison pour laquelle la taille de la population est un paramètre capital pour la performance des algorithmes génétiques [81].

En deuxième lieu, le Tableau 3.4 présente les résultats de l'AGP0 sans échanges avec la stratégie de réduction du nombre de générations (GEN) et le maintien de la taille de la population pour chaque processeur. En comparant ces résultats avec l'AG1, on remarque que seulement deux "Meilleur" ont été améliorés (PROB605 et PROB655) et aucun résultat "Médian" ou "Pire" n'a été amélioré. Les résultats de l'AGP0 avec la stratégie de réduction du nombre de générations ne permettent pas de conserver la qualité de solution obtenue par l'AG1. Contrairement à la précédente stratégie, les sous-populations disposent de tous leurs individus mais l'algorithme s'exécute sur un plus petit nombre de générations. Comme les sous-populations sont isolées, l'AGP0 se comporte un algorithme séquentiel limité à 500 générations sur 2 processeurs et à 250 générations sur 4 processeurs. On a déjà constaté dans la Section 3.2 que l'AG0 performait mieux avec 1000 générations au lieu de 300 et que la convergence moyenne se situait autour de 850 générations. Les résultats de l'AGP0 confirment donc ce constat. Pour appuyer davantage cette idée, la comparaison des résultats sur 2 et 4 processeurs montre bien l'effet de la diminution de moitié du nombre de générations. En effet, aucun résultat médian sur 4 processeurs n'est meilleur que sur 2 processeurs. On observe également une chute dans le nombre de problèmes où le résultat médian de l'AG1 est atteint (26 avec 2 processeurs contre 17 avec 4 processeurs).

Le Tableau 3.5 résume le nombre de résultats médians de l'AG1 atteints par l'exécution de l'AGP0 sur 2 et 4 processeurs avec chacune des deux stratégies en considérant un écart maximal de 5%. On observe que les performances sur 2 processeurs sont meilleures que sur 4 processeurs et que la stratégie GEN s'avère un peu plus performante que POP (respectivement 26/43 et 23/43). Ce phénomène peut être expliqué par le fait qu'en absence de communications, l'algorithme est plus efficace lorsqu'il dispose d'une population complète même s'il a moins de générations pour tenter d'améliorer la solution. C'est ce qu'a notamment constaté Cantù-

Prb	# comm.	B&B		AG1 Meilleur	AG1 Médian	AG1 Pire	Stratégie POP 2PROC Meilleur	2PROC Médian	2PROC Pire	4PROC Meilleur	4PROC Médian	4PROC Pire
PROB401	15	90	*	4,4	11,1	37,8	0,0	21,1	37,8	4,4	28,9	162,2
PROB403	15	3418	*	0,0	1,9	3,4	0,0	2,4	6,7	0,3	3,6	14,7
PROB404	15	1067	*	0,0	0,0	0,0	0,0	0,0	4,1	0,0	0,0	8,2
PROB407	15	1861	*	0,0	1,2	2,9	0,0	1,8	5,7	0,0	2,7	25,2
PROB408	15	5660	*	0,0	1,5	2,5	0,0	1,6	3,0	0,0	2,2	4,9
PROB501	25	264		3,0	4,9	12,1	1,9	6,1	31,1	3,4	10,6	129,5
PROB503	25	3511		-0,1	0,6	2,2	0,1	1,3	3,8	0,3	2,2	17,5
PROB507	25	7225	*	0,5	2,8	5,2	1,0	3,8	7,7	2,2	6,2	25,5
PROB508	25	2067		0,0	5,3	11,7	0,0	12,3	38,9	5,1	25,7	65,2
PROB601	35	30		43,3	140,0	236,7	93,3	193,3	346,7	120,0	266,7	716,7
PROB603	35	17774		1,4	3,2	4,6	2,8	4,5	7,7	3,0	6,1	11,8
PROB604	35	19277		1,3	2,7	4,7	2,2	4,1	7,0	3,4	6,3	12,0
PROB605	35	291		6,9	23,7	39,2	12,4	38,1	124,1	17,2	64,9	215,8
PROB607	35	13274		-0,3	3,1	9,2	1,6	4,3	10,0	3,8	8,4	18,4
PROB608	35	6704		-28,4	-23,0	-16,9	-27,6	-16,8	-4,3	-20,7	-10,2	6,4
PROB701	45	116		35,3	90,5	119,0	69,8	104,3	179,3	62,9	152,6	239,7
PROB703	45	27097		1,4	2,8	4,8	2,4	4,5	6,3	3,5	6,8	10,1
PROB704	45	15941		0,7	4,8	10,0	2,0	6,9	15,1	6,1	15,2	25,6
PROB705	45	234		45,3	77,8	106,0	49,6	89,7	227,8	66,7	103,0	215,0
PROB707	45	25070		-1,2	0,5	3,1	-0,9	3,4	6,7	1,4	6,2	10,4
PROB708	45	24123		-3,4	0,2	5,5	-0,7	3,4	9,0	3,7	8,9	15,5
PROB551	55	212		95,8	128,8	162,3	102,4	149,5	240,1	126,9	205,2	353,8
PROB553	55	40828		3,7	5,3	6,6	4,1	6,3	9,2	5,7	8,6	14,7
PROB554	55	15091		8,3	15,0	20,0	11,7	21,2	38,6	19,2	32,7	56,6
PROB557	55	36489		2,5	4,9	8,4	4,0	7,5	10,1	6,4	10,0	13,7
PROB558	55	20624		2,6	8,1	13,7	10,0	15,8	28,5	14,8	24,4	40,2
PROB651	65	295		72,2	112,2	168,5	90,8	128,8	171,9	103,1	161,7	228,8
PROB653	65	57779		5,3	6,3	10,0	5,7	8,3	10,2	6,8	9,9	13,2
PROB654	65	34468		7,6	11,3	15,7	11,9	18,0	24,1	15,4	25,2	32,5
PROB655	65	13		853,8	1107,7	1961,5	**838,5**	1369,2	**1892,3**	1115,4	1753,8	3192,3
PROB657	65	56246		4,0	5,4	6,8	5,5	7,9	10,2	7,0	10,1	13,8
PROB658	65	29308		1,9	6,8	11,9	7,2	12,9	20,3	11,6	25,1	35,3
PROB751	75	263		126,2	171,1	197,0	140,3	203,0	320,9	159,3	262,4	389,0
PROB753	75	78211		5,0	6,4	7,8	6,3	8,5	10,2	7,6	9,9	14,2
PROB754	75	35826		12,4	15,7	19,8	16,8	24,2	33,7	22,4	34,9	46,8
PROB757	75	61513		3,7	5,3	7,7	5,3	7,9	10,4	6,9	10,9	15,1
PROB758	75	40277		7,1	11,4	17,0	13,7	19,9	26,0	21,3	30,2	40,2
PROB851	85	453		123,4	164,2	191,2	150,3	177,3	233,3	148,8	237,1	341,9
PROB853	85	98540		4,9	7,0	9,0	6,4	7,8	10,4	7,9	10,7	13,7
PROB854	85	80693		7,4	9,3	12,1	8,9	13,4	16,2	12,0	17,9	23,2
PROB855	85	333		137,5	158,6	191,3	138,7	188,3	240,8	151,4	228,2	351,4
PROB857	85	89654		4,5	6,4	8,9	6,8	9,0	10,8	8,7	11,4	16,9
PROB858	85	77919		5,1	8,2	11,1	9,8	13,8	19,3	15,1	21,1	32,0

* Solution Optimale

Tableau 3.3 – Résultats AGP0 sans échanges avec stratégie POP

Prb	# comm.	B&B *	AG1 Meilleur	AG1 Médian	AG1 Pire	Stratégie GEN 2PROC Meilleur	2PROC Médian	2PROC Pire	4PROC Meilleur	4PROC Médian	4PROC Pire
PROB401	15	90 *	4,4	11,1	37,8	4,4	16,7	40,0	4,4	24,4	50,0
PROB403	15	3418 *	0,0	1,9	3,4	0,0	2,3	4,6	0,0	3,2	7,2
PROB404	15	1067 *	0,0	0,0	0,0	0,0	0,0	6,7	0,0	0,0	9,2
PROB407	15	1861 *	0,0	1,2	2,9	0,0	1,8	3,0	0,0	2,4	9,0
PROB408	15	5660 *	0,0	1,5	2,5	0,0	1,7	3,1	0,0	2,3	4,2
PROB501	25	264	3,0	4,9	12,1	1,5	4,9	12,9	2,7	8,3	15,2
PROB503	25	3511	-0,1	0,6	2,2	0,0	1,2	3,8	0,4	1,9	5,0
PROB507	25	7225 *	0,5	2,8	5,2	1,5	4,4	7,7	1,9	5,3	11,1
PROB508	25	2067	0,0	5,3	11,7	0,0	6,7	24,4	4,2	20,2	49,6
PROB601	35	30	43,3	140,0	236,7	66,7	180,0	346,7	96,7	243,3	376,7
PROB603	35	17774	1,4	3,2	4,6	1,9	4,0	6,5	2,8	5,6	8,9
PROB604	35	19277	1,3	2,7	4,7	1,5	3,7	6,3	3,1	5,6	9,8
PROB605	35	291	6,9	23,7	39,2	**-0,7**	38,1	114,1	16,5	59,8	113,1
PROB607	35	13274	-0,3	3,1	9,2	2,5	4,6	9,1	3,2	8,0	17,6
PROB608	35	6704	-28,4	-23,0	-16,9	-25,6	-18,2	-5,3	-23,6	-11,0	6,2
PROB701	45	116	35,3	90,5	119,0	69,8	112,9	186,2	88,8	144,0	245,7
PROB703	45	27097	1,4	2,8	4,8	2,3	4,5	7,9	3,3	6,5	9,8
PROB704	45	15941	0,7	4,8	10,0	1,7	8,5	13,4	4,7	12,8	29,2
PROB705	45	234	45,3	77,8	106,0	60,7	94,0	120,1	70,9	106,0	161,5
PROB707	45	25070	-1,2	0,5	3,1	-0,5	3,5	8,6	1,7	5,7	10,3
PROB708	45	24123	-3,4	0,2	5,5	-0,2	3,2	9,2	2,2	6,6	11,8
PROB551	55	212	95,8	128,8	162,3	119,3	163,2	255,7	153,3	223,1	357,5
PROB553	55	40828	3,7	5,3	6,6	3,9	7,2	8,7	5,8	8,6	11,2
PROB554	55	15091	8,3	15,0	20,0	10,3	20,9	28,6	15,8	28,9	41,8
PROB557	55	36489	2,5	4,9	8,4	5,0	7,5	10,3	5,8	9,0	14,2
PROB558	55	20624	2,6	8,1	13,7	6,2	14,4	24,8	10,4	22,4	35,6
PROB651	65	295	72,2	112,2	168,5	96,3	132.2	172,5	116,9	173,6	272,5
PROB653	65	57779	5,3	6,3	10,0	6,1	8,4	10,0	6,5	10,7	13,2
PROB654	65	34468	7,6	11,3	15,7	12,2	16,1	22,2	15,6	22,8	30,8
PROB655	65	13	853,8	1107,7	1961,5	692.3	1469,2	2169,2	1169,2	1846,2	3507,7
PROB657	65	56246	4,0	5,4	6,8	4,9	7,2	10,5	6,3	10,1	14,4
PROB658	65	29308	1,9	6,8	11,9	6,7	12,3	23,8	9,3	20,9	34,6
PROB751	75	263	126,2	171,1	197,0	160,5	211,4	319,4	185,2	304,2	433,5
PROB753	75	78211	5,0	6,4	7,8	6,1	8,0	11,6	7,6	10,5	14,4
PROB754	75	35826	12,4	15,7	19,8	16,7	23,9	31,8	22,1	31,6	43,5
PROB757	75	61513	3,7	5,3	7,7	5,7	8,1	11,3	7,4	11,1	16,0
PROB758	75	40277	7,1	11,4	17,0	12,3	19,1	34,0	18,5	26,3	40,3
PROB851	85	453	123,4	164,2	191,2	139,7	192,9	287,2	193,8	271,3	429,8
PROB853	85	98540	4,9	7,0	9,0	6,0	8,7	11,0	8,4	11,2	14,6
PROB854	85	80693	7,4	9,3	12,1	9,5	12,7	16,6	11,2	17,5	24,1
PROB855	85	333	137,5	158,6	191,3	144,1	194,9	252.0	196,1	265,2	340,8
PROB857	85	89654	4,5	6,4	8,9	6,1	9,1	12,6	8,5	12,2	15,8
PROB858	85	77919	5,1	8,2	11,1	8,7	14,0	21,0	14,4	19,7	26,4

* Solution Optimale

Tableau 3.4 – Résultats AGP0 sans échanges avec stratégie GEN

Paz [27] [28] en analysant le comportement des algorithmes génétiques parallèles possédant des sous-populations isolées.

Stratégie	2 Processeurs	4 Processeurs
POP	23	14
GEN	26	17

Tableau 3.5 – Nombre de résultats médians de l'AG1 atteints sans communication par l'AGP0

Pour les deux stratégies, les faibles performances de l'AGP0 sans communication s'expliquent par la diminution de l'espace de recherche et par l'absence de mécanismes de conservation de la diversité. Cette performance est prévisible puisque, privé de communication, l'AGP0 se comporte comme un algorithme séquentiel mais disposant d'un espace de recherche plus restreint que AG1.

Le Tableau 3.6 présente les temps d'exécutions *(T)* et les accélérations *(A)* relatifs aux essais numériques précédents. Les temps d'exécution représentent les moyennes des 20 exécutions. On constate que l'AGP0 sans communication produit de très bons temps d'exécution et des accélérations qui, pour la plupart des problèmes, atteignent le maximum théorique. C'est ce qu'on a appelé une accélération linéaire à la Section 2.1.5.1.

Deux problèmes tests de tailles respectives de 250 et 500 commandes ont été ajoutés afin d'observer le comportement de l'AGP0 lorsque la taille du problème augmente. De ce fait, le comportement de l'algorithme pourra également être analysé sur la base de l'utilisation d'un plus grand nombre de processeurs.

Les Tableaux 3.7 et 3.8 présentent les résultats pour ces deux problèmes en utilisant 2, 4, 8, 16 et 32 processeurs avec les deux stratégies POP et GEN sans échanges d'information. Comme ces problèmes ne sont pas des problèmes de la littérature et qu'on ne dispose pas de l'optimum trouvé par une méthode exacte, on suppose que l'optimum est le meilleur résultat trouvé par l'AG1 pour fins de comparaison. En premier lieu, le Tableau 3.7 présente des résultats similaires observés pour les problèmes de petite taille, à savoir que la qualité de solution se dégrade considérablement avec l'augmentation du nombre de processeurs avec la stratégie de subdivision de la population (POP).

Au Tableau 3.8, avec la stratégie de réduction du nombre de générations et de maintien

		AG1	AGP0 POP				AGP0 GEN			
			2PROC		4PROC		2PROC		4PROC	
Prb	#comm.	T	T	A	T	A	T	A	T	A
PROB401	15	0,318	0,172	1,8	0,106	3,0	0,158	2,0	0,080	4,0
PROB403	15	0,305	0,149	2,1	0,099	3,1	0,150	2,0	0,076	4,0
PROB404	15	0,347	0,190	1,8	0,125	2,8	0,173	2,0	0,087	4,0
PROB407	15	0,327	0,179	1,8	0,114	2,9	0,173	1,9	0,087	3,7
PROB408	15	0,350	0,189	1,9	0,124	2,8	0,175	2,0	0,087	4,0
PROB501	25	0,834	0,422	2,0	0,217	3,8	0,445	1,9	0,242	3,5
PROB503	25	0,889	0,471	1,9	0,220	4,0	0,461	1,9	0,237	3,8
PROB507	25	1,011	0,531	1,9	0,273	3,7	0,510	2,0	0,254	4,0
PROB508	25	1,074	0,554	1,9	0,292	3,7	0,541	2,0	0,270	4,0
PROB601	35	1,999	0,997	2,0	0,487	4,1	0,984	2,0	0,499	4,0
PROB603	35	1,930	0,981	2,0	0,483	4,0	0,966	2,0	0,482	4,0
PROB604	35	2,015	1,052	1,9	0,527	3,8	1,009	2,0	0,507	4,0
PROB605	35	1,964	0,998	2,0	0,485	4,0	1,005	2,0	0,501	3,9
PROB607	35	1,974	1,028	1,9	0,513	3,9	1,019	1,9	0,508	3,9
PROB608	35	2,214	1,153	1,9	0,571	3,9	1,108	2,0	0,557	4,0
PROB701	45	3,389	1,711	2,0	0,900	3,8	1,664	2,0	0,851	4,0
PROB703	45	3,294	1,665	2,0	0,872	3,8	1,660	2,0	0,824	4,0
PROB704	45	3,516	1,781	2,0	0,888	4,0	1,764	2,0	0,897	3,9
PROB705	45	3,435	1,712	2,0	0,916	3,7	1,720	2,0	0,861	4,0
PROB707	45	3,470	1,749	2,0	0,908	3,8	1,718	2,0	0,870	4,0
PROB708	45	3,674	1,843	2,0	0,921	4,0	1,845	2,0	0,922	4,0
PROB551	55	5,129	2,587	2,0	1,335	3,8	2,615	2,0	1,313	3,9
PROB553	55	5,135	2,625	2,0	1,333	3,9	2,620	2,0	1,304	3,9
PROB554	55	5,482	2,807	2,0	1,477	3,7	2,758	2,0	1,391	3,9
PROB557	55	5,441	2,786	2,0	1,470	3,7	2,743	2,0	1,379	3,9
PROB558	55	5,745	2,941	2,0	1,554	3,7	2,882	2,0	1,460	3,9
PROB651	65	6,950	3,796	1,8	2,189	3,2	3,769	1,8	1,892	3,7
PROB653	65	7,426	3,841	1,9	2,162	3,4	3,783	2,0	1,893	3,9
PROB654	65	7,974	3,983	2,0	2,222	3,6	4,023	2,0	2,032	3,9
PROB655	65	7,528	3,849	2,0	2,252	3,3	3,880	1,9	1,953	3,9
PROB657	65	7,895	4,009	2,0	2,246	3,5	3,946	2,0	1,979	4,0
PROB658	65	8,383	4,166	2,0	2,319	3,6	4,218	2,0	2,127	3,9
PROB751	75	10,321	5,206	2,0	2,811	3,7	5,203	2,0	2,631	3,9
PROB753	75	10,564	5,315	2,0	2,803	3,8	5,277	2,0	2,646	4,0
PROB754	75	11,090	5,664	2,0	2,994	3,7	5,601	2,0	2,844	3,9
PROB757	75	10,989	5,602	2,0	2,961	3,7	5,511	2,0	2,764	4,0
PROB758	75	11,438	5,817	2,0	3,083	3,7	5,758	2,0	2,908	3,9
PROB851	85	13,401	6,868	2,0	3,487	3,8	6,837	2,0	3,477	3,9
PROB853	85	13,932	6,951	2,0	3,483	4,0	6,976	2,0	3,483	4,0
PROB854	85	14,689	7,382	2,0	3,711	4,0	7,409	2,0	3,757	3,9
PROB855	85	13,853	7,091	2,0	3,583	3,9	7,056	2,0	3,591	3,9
PROB857	85	14,371	7,202	2,0	3,616	4,0	7,204	2,0	3,620	4,0
PROB858	85	15,241	7,632	2,0	3,834	4,0	7,899	1,9	3,905	3,9

Tableau 3.6 – Acccélerations AGP0 sans échanges

Prb	# comm.	Meilleur AG0	AG1			AGP0 POP					
						2 PROC			4 PROC		
			Meilleur	Médian	Pire	Meilleur	Médian	Pire	Meilleur	Médian	Pire
PREX250	250	504875	0,0	4,8	8,2	10,7	15,4	21,9	22,6	29,6	41,4
PREX500	500	38537	0,0	24,9	53,6	26,5	51,0	90,7	34,7	79,4	137,6
Prb	# comm.	Meilleur AG0	AGP0 POP								
			8 PROC			16 PROC			32 PROC		
			Meilleur	Médian	Pire	Meilleur	Médian	Pire	Meilleur	Médian	Pire
PREX250	250	504875	35,6	29,6	41,4	50,5	67,9	85,3	70,8	86,5	102,9
PREX500	500	38537	91,4	79,4	137,6	164,2	233,7	332,4	252,8	349,9	526,7

Tableau 3.7 – Résultats AGP0 sans échanges avec stratégie POP sur les grands problèmes

de la taille de la population pour chaque processeur (GEN), on observe, encore une fois, que la qualité de la solution se dégrade considérablement avec l'augmentation du nombre de processeurs. Cette dégradation de la qualité est toutefois plus rapide qu'avec la stratégie POP, ce qui est l'inverse de ce qui avait été observé pour les plus petits problèmes.

Prb	# comm.	Meilleur AG0	AG1			AGP0 GEN					
						2 PROC			4 PROC		
			Meilleur	Médian	Pire	Meilleur	Médian	Pire	Meilleur	Médian	Pire
PREX250	250	504875	0,0	4,8	8,2	12,5	18,0	26,2	31,1	39,3	50,2
PREX500	500	38537	0,0	24,9	53,6	97,9	135,0	160,4	197,3	260,1	306,0
Prb	# comm.	Meilleur AG0	AGP0 GEN								
			8 PROC			16 PROC			32 PROC		
			Meilleur	Médian	Pire	Meilleur	Médian	Pire	Meilleur	Médian	Pire
PREX250	250	504875	55,8	66,5	80,4	77,7	92,5	104,0	100,4	115,8	126,9
PREX500	500	38537	361,2	425,8	485,9	476,0	575.5	642,4	593,3	688,8	745,0

Tableau 3.8 – Résultats AGP0 sans échanges avec stratégie GEN sur les grands problèmes

Le Tableau 3.9 présente les accélérations obtenues sur ces deux problèmes. Comme c'était le cas précédemment, les accélérations sont très bonnes et l'augmentation du nombre de processeurs permet de mieux distinguer la performance des deux stratégies. En effet, les accélérations obtenues avec la stratégie GEN sont, dans tous les cas, inférieures à la stratégie POP. La non-linéarité des accélérations de la stratégie GEN s'explique par la conservation du nombre d'individus sur chaque processeur. En effet, il s'agit de la série de tests la plus exigeante en terme de calculs avec des sous-populations complètes (250 et 500 individus respectivement) à gérer. Cet écart n'était pas observable dans les problèmes de petite taille.

Cette première série de tests sur l'algorithme AGP0 sans échanges d'information permet de conclure que cette forme de parallélisation produit des résultats peu performants

		AG1	AGP0 POP									
			2PROC		4PROC		8PROC		16PROC		32PROC	
Prb	#comm.	T	T	A	T	A	T	A	T	A	T	A
PREX250	250	188,862	94,028	2,0	47,803	4,0	23,744	8,0	11,992	15,7	6,096	31,0
PREX500	500	1091,823	544,057	2,0	273,157	4,0	137,717	7,9	69,527	15,7	35,176	31,0
		AG1	AGP0 GEN									
			2PROC		4PROC		8PROC		16PROC		32PROC	
Prb	# comm.	T	T	A	T	A	T	A	T	A	T	A
PREX250	250	188,862	96,934	1,9	51,106	3,7	26,647	7,1	13,830	13,7	7,139	26,5
PREX500	500	1091,823	577,199	1,9	295,745	3,7	151,083	7,2	78,651	13,9	40,855	26,7

Tableau 3.9 – Accélérations sans échanges sur les grands problèmes

en ce qui concerne la qualité des solutions pour les deux stratégies testées en comparaison avec l'AG1. Cependant, pour les problèmes de 15 à 85 commandes, la stratégie GEN a donné de meilleurs résultats que la stratégie POP et cela s'est inversé dans le cas des problèmes de taille 250 et 500 commandes. Du point de vue des accélérations, on a globalement constaté des accélérations linéaires sauf dans le cas des gros problèmes avec la stratégie de réduction des générations. Ainsi, en poussant les tests sur les gros problèmes avec un plus grand nombre de processeurs, on a mesuré l'impact des deux stratégies sur les accélérations. D'après la littérature, les communications représentent le principal frein aux accélérations et c'est cet impact que l'on va étudié dans la prochaine section.

3.4.2 AG avec échange d'information

Le plan d'expérience présenté précédemment fait intervenir deux nouveaux paramètres dans le cas où il existe des échanges d'information entre les processeurs. Le premier paramètre concerne l'intervalle I, c'est-à-dire le nombre de générations entre deux échanges d'information entre deux processeurs. Le deuxième paramètre précise le taux de migration R, c'est-à-dire le pourcentage de la population qui est transmise à l'autre processeur. Les essais numériques ont été effectués en faisant varier ces deux paramètres selon la stratégie choisie (GEN ou POP). Les intervalles de migration testés sont à toutes les 1, 5, 10, 20, 50, 100 et 500 générations et les taux de migration sont de 10%, 20% et 50% de la population du processeur. Ces 21 scénarios sont évalués sur l'ensemble des problèmes.

Pour l'implémentation des échanges, les principales fonctions MPI utilisées sont :

MPI_Wtime pour le chronométrage des différentes itérations, MPI_Gather pour le rassemblement des résultats obtenus par chaque processeur, MPI_Send et MPI_Recv pour représenter les communications en anneau entre les processeurs avec le même principe décrit dans la Figure 2.9.

Le Tableau 3.10 résume la performance des différents scénarios selon la stratégie de subdivision de la population (POP). Pour chaque scénario, on a établi le nombre de problèmes dont les résultats médians sont supérieurs (>), équivalent (=) ou pires (<) que les résultats obtenus avec l'AG1. Une tolérance de 5% est encore utilisée pour cette comparaison de performance. D'après les résultats de ce tableau, on peut établir que les scénarios permettant d'améliorer ou de conserver le plus grand nombre de problèmes (résultats présentés en caractères gras) au niveau de la qualité des résultats se ressemblent selon le nombre de processeurs. En effet, les scénarios extrêmes quant à la fréquence d'échanges semblent moins performants et on note également que la fréquence d'échanges doit diminuer avec l'augmentation du nombre de processeurs. À ce qui a trait au taux de migration, il semble préférable de l'augmenter avec l'augmentation du nombre de processeurs. On remarque également, pour les scénarios performants, que le passage de 2 à 4 processeurs fait diminuer le nombre de problèmes améliorés ou atteints. C'est le même constat qui avait été observé à la Section 3.4.1 pour l'AGP0 sans communication et dont les performances sont résumées dans la dernière ligne du Tableau 3.10.

En réalisant une analyse plus détaillée de l'ensemble des résultats, on remarque également que, pour les fréquences 10 et 20, l'amélioration des problèmes concerne surtout les problèmes de taille supérieure à 25. Les améliorations ont été rapportées principalement sur les problèmes ayant les caractéristiques PTV=Low, TF=Low et DDR=Narrow (correspond aux problèmes dont le numéro se termine par 1) et également sur les problèmes ayant les caractéristiques PTV=High, TF=Low et DDR=Narrow (correspond aux problèmes dont le numéro se termine par un 5). En effet, pour chaque groupe de problèmes de taille supérieure à 25 commandes, tous les problèmes ayant ces caractéristiques ont été améliorés.

Sur la base des résultats du Tableau 3.10, on peut donc établir que les échanges entre les processeurs sont généralement bénéfiques pour l'amélioration de la qualité des résultats. Un des facteurs majeurs affectant la qualité de solution dans un algorithme génétique est la

diversité de la population. Le maintien de cette diversité, malgré la diminution de la taille des sous-populations, tout au long de l'exécution de la version parallèle est assuré par les échanges d'individus [70]. Les algorithmes coopèrent et se diversifient les uns les autres pour faire émerger de meilleures solutions.

Scénario	Stratégie POP					
	2 PROC			2 PROC		
	>	=	<	>	=	<
I1R10	1	34	8	0	22	21
I1R20	4	30	9	1	29	13
I1R50	0	20	23	0	19	24
I5R10	**7**	**35**	1	6	30	7
I5R20	**7**	**34**	2	**5**	**35**	3
I5R50	**4**	**38**	1	**5**	**36**	2
I10R10	**8**	**33**	2	8	28	7
I10R20	**8**	**35**	0	**7**	**33**	3
I10R50	**8**	**34**	1	**7**	**33**	3
I20R10	**9**	**33**	1	6	31	6
I20R20	**6**	**37**	0	**6**	**34**	3
I20R50	**9**	**34**	0	**7**	**35**	1
I50R10	**7**	**35**	1	4	30	9
I50R20	**7**	**35**	1	**4**	**36**	3
I50R50	**7**	**35**	1	**3**	**39**	1
I100R10	**5**	**35**	1	0	39	4
I100R20	**5**	**38**	0	1	36	6
I100R50	**7**	**36**	0	1	35	7
I500R10	1	36	6	0	18	25
I500R20	1	34	8	0	19	24
I500R50	1	36	6	0	21	22
NoCom	0	23	20	0	14	29

Tableau 3.10 – Résultats AGP0 avec échanges (POP)

Le Tableau 3.11 présente la performance des différents scénarios selon la stratégie de réduction du nombre de générations (GEN). Les scénarios permettant d'améliorer ou de conserver le plus grand nombre de résultats médians de l'AG1 sont présentés en caractères gras. On peut noter que l'échange à toutes les générations ne semble pas très performant et qu'il ne faut pas, non plus, trop espacer l'échange compte tenu de la réduction du nombre de générations. L'augmentation du nombre de processeurs entraîne également l'utilisation d'un échange plus fréquent pour obtenir de bons résultats. Pour les scénarios avec les fréquences d'échanges produisant les meilleurs résultats, on note que le taux de migration n'a pas beaucoup d'influence. Toutefois, le passage de 2 à 4 processeurs fait chuter dramatiquement le nombre de

problèmes dont les résultats médians de l'AG1 sont améliorés. En effet, seul le scénario I5R50 avec l'utilisation de 4 processeurs permet d'améliorer un problème. Les différents scénarios permettent, dans l'ensemble, d'obtenir des résultats de meilleure qualité que la version sans communication.

Globalement, les essais numériques réalisés selon les deux stratégies permettent ainsi d'établir que la stratégie POP s'avère supérieure à la stratégie GEN pour l'amélioration des résultats médians pour l'ensemble des problèmes. Compte tenu de la taille des problèmes, il semble également préférable de limiter le nombre de processeurs à 2.

Scénario	Stratégie GEN					
	2 PROC			2 PROC		
	>	=	<	>	=	<
I1R10	1	35	7	0	32	11
I1R20	1	31	11	0	31	12
I1R50	0	21	22	0	21	22
I5R10	**4**	**37**	2	**0**	**36**	7
I5R20	**4**	**40**	0	**0**	**37**	6
I5R50	**5**	**36**	2	**1**	**38**	4
I10R10	**5**	**36**	2	**0**	**36**	7
I10R20	**6**	**36**	1	**0**	**37**	6
I10R50	**3**	**37**	3	**0**	**37**	6
I20R10	**5**	**36**	2	0	32	11
I20R20	**4**	**37**	2	0	35	8
I20R50	**4**	**38**	1	0	34	9
I50R10	**1**	**39**	3	0	28	15
I50R20	2	37	4	0	28	15
I50R50	2	36	5	0	28	15
I100R10	3	35	5	0	26	17
I100R20	1	38	4	0	25	18
I100R50	2	35	6	0	24	19
I500R10	0	35	8	0	17	26
I500R20	0	33	10	0	16	27
I500R50	0	33	10	0	16	24
NoCom	0	26	17	0	17	26

Tableau 3.11 – Résultats AGP0 avec échanges (GEN)

Du point de vue de l'accélération, le graphique 3.2 illustre l'impact des communications sur le temps d'exécution de l'algorithme. On constate que, pour 2 et 4 processeurs, la stratégie de réduction des générations (GEN) s'avère plus rapide que celle par subdivision de la population (POP), quelque soit le scénario. Les fréquences d'échanges qui, dans les tableaux précédents, ont donné les meilleurs résultats possèdent des accélérations intéressantes. L'aspect croissant

des courbes de gauche à droite est la conséquence directe de la fréquence des échanges. Dans la partie gauche du graphe, les échanges s'effectuent à toutes les générations et sont de moins en moins fréquents lorsqu'on progresse sur l'axe des abscisses. Cependant, l'augmentation progressive de l'accélération est aussi synonyme de diminution de la qualité comme on l'a montré précédemment. Paradoxalement, lorsque il y trop d'échanges, l'accélération et la qualité sont toutes les deux affectées puisque les algorithmes sur chaque processeur n'ont pas le temps de travailler dans leur espace de recherche respectif et toutes les sous-populations vont finir par se ressembler [28].

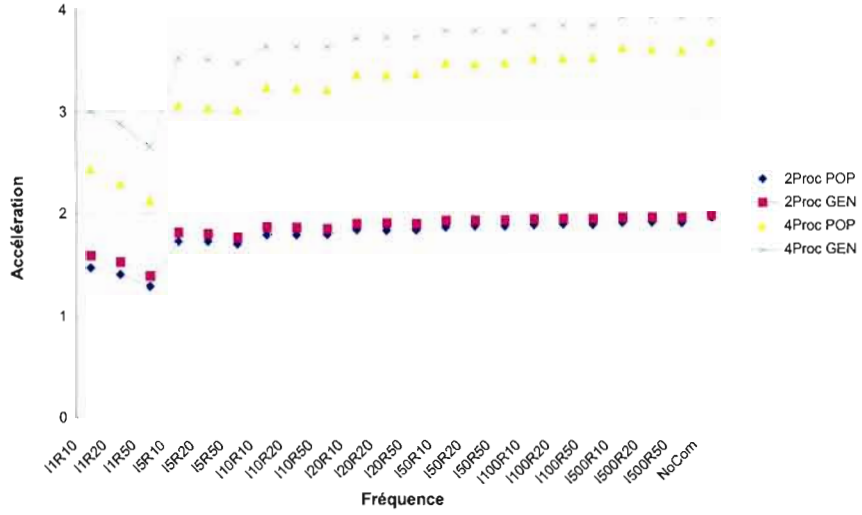

Figure 3.2 – Accélérations des petits problèmes sur 2 et 4 processeurs

Examinons maintenant la performance de l'AGP0 avec les 21 scénarios pour les problèmes de 250 et 500 commandes avec un plus grand nombre de processeurs. Les résultats présentés aux Tableaux 3.12 et 3.13 expriment l'écart de la solution médiane trouvée par rapport au résultat médian l'AG1, ceci avec la stratégie POP.

Au Tableau 3.12, on constate, tout d'abord, qu'aucun scénario ne permet d'améliorer la qualité de solution de l'AG1 et que peu d'entre eux réussissent à la conserver. En utilisant une marge de 5%, seuls les résultats indiqués en caractères gras au Tableau 3.12 expriment une qualité de solution équivalente à l'AG1. Les meilleurs résultats obtenus avec chaque groupe de processeurs sont soulignés. On note ainsi que les meilleurs résultats tendent à apparaître pour des fréquences plus petites avec l'augmentation du nombre de processeurs. Globalement,

la meilleure performance est obtenue en utilisant seulement 2 processeurs avec des fréquences d'échanges entre 5 et 20 et sans vraiment d'impact sur le choix du taux de migration.

Avec le problème de 500 commandes dont les résultats sont présentés au Tableau 3.13, il est possible d'améliorer la solution trouvée par l'AG1 avec certains scénarios jusqu'à 16 processeurs. Les meilleurs résultats obtenus avec chaque groupe de processeurs sont soulignés. Plusieurs scénarios permettent également de conserver un niveau de qualité équivalent à l'AG1. Ces résultats sont indiqués en caractères gras au Tableau 3.13. Tout comme le problème de 250 commandes, on note ainsi que les meilleurs résultats tendent à apparaître pour des fréquences plus petites avec l'augmentation du nombre de processeurs. L'analyse des problèmes de 250 et 500 commandes nous amène à un constat différent par rapport aux petits problèmes, à savoir que l'augmentation du nombre de processeurs n'est pas forcement néfaste à la qualité, et selon la fréquence, on a pu atteindre de très bons résultats. On ne peut cependant tirer de conclusion quant à la fréquence idéale à utiliser pour ce genre de problèmes. L'apport des communications semble améliorer la qualité des résultats mais sans tendance particulière.

	Stratégie POP				
	PREX 250				
Scénario	2 PROC	4 PROC	8 PROC	16 PROC	32 PROC
I1R10	10,4	10,7	12,5	21,4	86,3
I1R20	12,5	10,5	10,2	11,6	28,6
I1R50	19,5	16,1	15,8	14,1	17,7
I5R10	5.7	6,7	7,1	13,4	86,3
I5R20	5,3	6,0	**4,6**	11,4	21,9
I5R50	**4,9**	**4,0**	6,1	10,3	22,6
I10R10	**5,0**	7,0	8,3	18.0	86,7
I10R20	**4,4**	5,3	8.1	15,3	32,2
I10R50	**4,3**	5,7	7,9	17,6	31,9
I20R10	6,8	7,1	13,3	26,9	86,4
I20R20	**4,5**	6,0	12,3	24,5	43,3
I20R50	**4,3**	7,4	12,9	24,2	44,4
I50R10	5,6	11,6	21,8	38,1	86,4
I50R20	6,3	10,7	21,0	38,4	58,5
I50R50	5,4	11.2	23,2	39.6	60,3
I100R10	8,1	15.8	29,1	48,2	86.4
I100R20	7.5	16,2	29,6	47.5	68,2
I100R50	7,9	16.6	29,1	49.5	69,6
I500R10	12,2	26.0	41,6	62.0	86,7
I500R20	13,7	25,0	41.8	62,6	81,1
I500R50	11,5	24,7	43,1	62,6	81,0
NoCom	15,4	29,6	47,6	67,9	86,5

Tableau 3.12 – Résultats AGP0 sur le problème 250 avec échanges (POP)

Il est également intéressant de constater la différence entre les deux problèmes 250

et 500 dans les fréquences qui donnent les meilleurs résultats. Un anneau constitué de 4 processeurs a produit les meilleurs résultats médians de la série de tests du problème 250, alors qu'il a fallu un anneau de 16 processeurs pour le problème 500.

	Stratégie POP				
	PREX 500				
Scénario	2 PROC	4 PROC	8 PROC	16 PROC	32 PROC
I1R10	13,7	9,6	**3,0**	5,5	61,8
I1R20	13,7	**-1,0**	**-4,3**	**-5,9**	**12,7**
I1R50	39,6	13,5	**2,2**	**4,3**	19,8
I5R10	5,5	4,1	9,1	24,0	63,9
I5R20	**1,5**	**-2,0**	**3,5**	13,8	46,7
I5R50	**2,4**	**-5,5**	**-5,1**	9,3	49,0
I10R10	**4,6**	10.0	23,1	49,8	100,6
I10R20	**4,2**	6,2	16,6	39,9	88,7
I10R50	8,1	**1,9**	10,5	31,9	85,3
I20R10	13,5	19,6	43.0	79,9	151,7
I20R20	9,0	17,8	31,3	74,8	135,5
I20R50	10,8	12,5	32,1	66,3	132,2
I50R10	20,3	36,6	68,0	124,6	215,1
I50R20	22,5	32.6	64,4	116,0	205,0
I50R50	11,4	28,1	65,4	117.5	207,1
I100R10	22,5	48,5	87,6	156,6	256,3
I100R20	27,7	45.9	85,9	155,5	250,1
I100R50	22.3	50,4	87,8	156,3	253,4
I500R10	35,7	68,4	123,2	203,2	315,5
I500R20	42,3	70,9	124,6	205,6	315,0
I500R50	39,0	69,0	124,1	209,3	316,3
NoCom	51,0	79.4	140,4	233,7	349,9

Tableau 3.13 - Résultats AGP0 sur le problème 500 avec échanges (POP)

Du point de vue de l'accélération, on remarque d'après les graphiques 3.3 et 3.4 que les fréquences qui ont donné les meilleurs résultats au niveau de la qualité sont celles avec les accélérations les moins bonnes. Globalement, on constate un coût de communication important lorsque les échanges sont importants et les accélérations linéaires ne sont pas aussi fréquentes que pour les petits problèmes. L'aspect croissant des courbes est maintenu avec un net décalage vers le bas lorsqu'on compare les accélérations de 250 et 500. Le fait que la taille du problème est multipliée par 2 fait augmenter la taille des messages échangés entre les processeurs et, par conséquent, fait augmenter les coûts de communications qui freinent l'accélération.

Pour terminer le plan d'expérience, examinons les résultats des grands problèmes avec la stratégie GEN. Les Tableaux 3.14 et 3.15 présentent les écarts des résultats médians de AGP0 par rapport à l'optimum trouvé par AG1. On constate d'après ces tableaux que les résultats obtenus ne sont pas de la même qualité que la précédente stratégie. En considérant

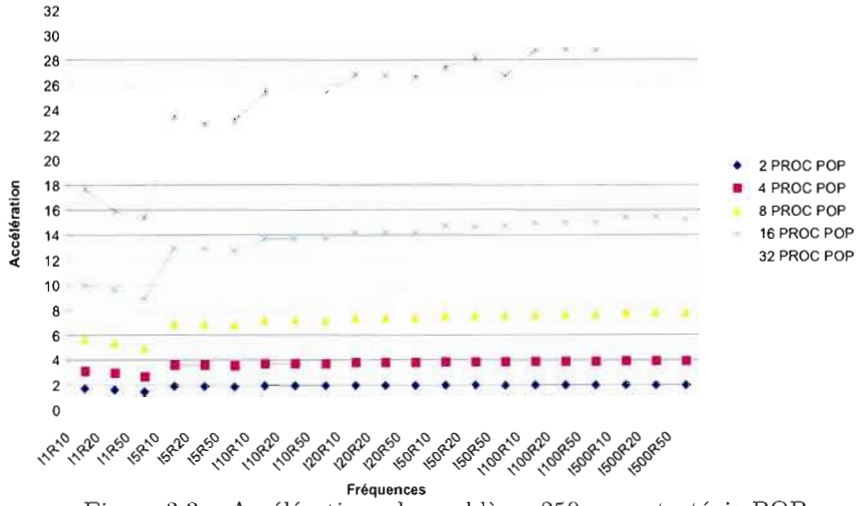

Figure 3.3 – Accélérations du problème 250 avec stratégie POP

la même marge de 5%, aucun scénario n'obtient un résultat équivalent à AG1. Les meilleurs résultats par nombre de processeur sont soulignés et cet écart grandit avec la taille de l'anneau.

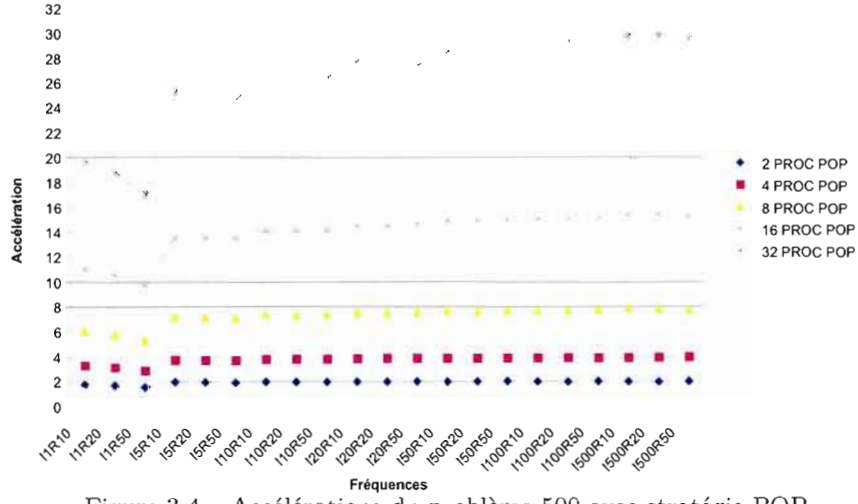

Figure 3.4 – Accélérations du problème 500 avec stratégie POP

En comparant les accélérations obtenues avec cette stratégie, on remarque le maintien de la tendance croissante en diminuant la fréquence des échanges. De plus, les accélérations illustrées dans les Figures 3.5 et 3.6 sont plus regroupées vers une moyenne et pour un nombre de processeurs supérieur à 4 sont toutes sous-linéaires, autant pour le problème 250 que 500.

	Stratégie GEN				
	PREX 250				
Scénario	2 PROC	4 PROC	8 PROC	16 PROC	32 PROC
I1R10	15,0	23,7	40,5	66,7	96,6
I1R20	17,5	21,6	37,7	63,9	94,8
I1R50	25,6	28,7	42,3	65,9	96,0
I5R10	9,9	22,9	48,5	81,0	108,6
I5R20	10,2	22,3	46,3	78,6	107,0
I5R50	9,3	20,1	46,2	78,3	106,8
I10R10	10,4	25,1	54,8	85,3	111,1
I10R20	8,2	24,6	52,9	83,8	110,1
I10R50	8,7	23,7	52.0	83,4	110,2
I20R10	9,2	28,8	57,8	88,0	114.0
I20R20	8,0	28,7	57,9	87,8	114,0
I20R50	8,4	27,4	56.5	86,5	113,2
I50R10	11,7	32,0	62,4	90,8	115,3
I50R20	11,4	30,9	61,7	91,0	115,4
I50R50	10,7	31,4	61,9	90.3	115,6
I100R10	11,5	35,6	62,8	93,0	115,5
I100R20	12,1	35,1	63,2	92.5	115,3
I100R50	14,0	35,1	63,5	92.0	115,4
I500R10	16,6	38,9	65,8	92,9	115,6
I500R20	15,9	38.8	66,3	92,9	115.8
I500R50	16,2	39,7	65,9	93.3	115,6
NoCom	18,0	39,3	66,5	92,5	115,8

Tableau 3.14 – Résultats AGP0 sur le problème 250 avec échanges (GEN)

	Stratégie GEN				
	PREX 500				
Scénario	2 PROC	4 PROC	8 PROC	16 PROC	32 PROC
I1R10	83,4	151,9	256,0	408,5	575,9
I1R20	78.8	156,8	247.7	402,2	573,6
I1R50	123,6	197,9	276,7	431,2	582.3
I5R10	78,0	168,3	316,1	493.2	640,5
I5R20	71,6	158,5	300,1	492,6	639.1
I5R50	73,1	154,6	304,6	485,5	633,6
I10R10	80.2	184,7	346.3	526,5	660,7
I10R20	75,6	178,1	342,2	518.3	658.1
I10R50	77,4	173,8	343,9	518,1	656,4
I20R10	100.8	206,5	377,7	547,5	676,3
I20R20	96,1	202,5	372,6	544.8	679.3
I20R50	87.2	198,6	372,2	542.6	676,1
I50R10	112,7	227,9	403,2	559,1	688,7
I50R20	104.5	223,3	399,7	562.6	688,9
I50R50	107.6	222,9	399,5	557,6	689,8
I100R10	112,0	243,2	411.4	573,9	687,1
I100R20	116,4	234,1	416,2	575,4	687.1
I100R50	111,5	241,5	408,0	574,7	686,4
I500R10	126,5	256,1	423,8	573,8	687,7
I500R20	120,1	260,7	422.6	571,3	688.7
I500R50	123,2	257,5	428,7	576,1	687,6
NoCom	135.0	260,1	425,8	575,5	688,8

Tableau 3.15 – Résultats AGP0 sur le problème 500 avec échanges (GEN)

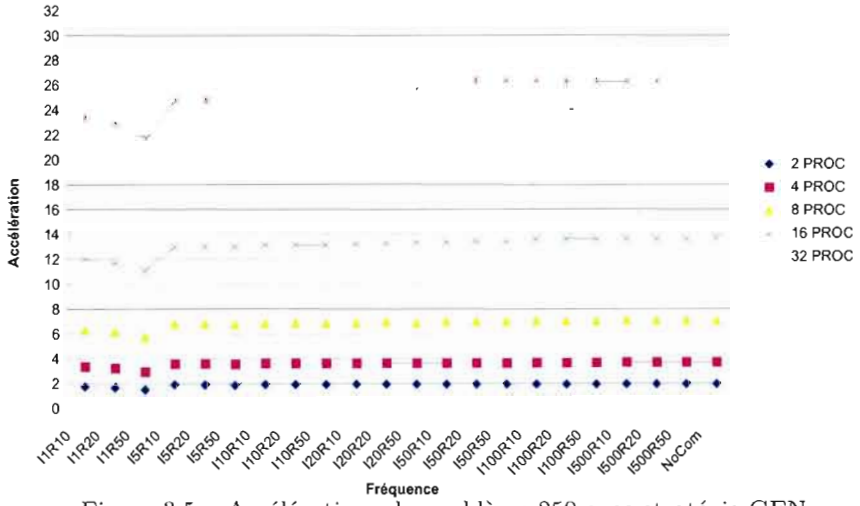

Figure 3.5 – Accélérations du problème 250 avec stratégie GEN

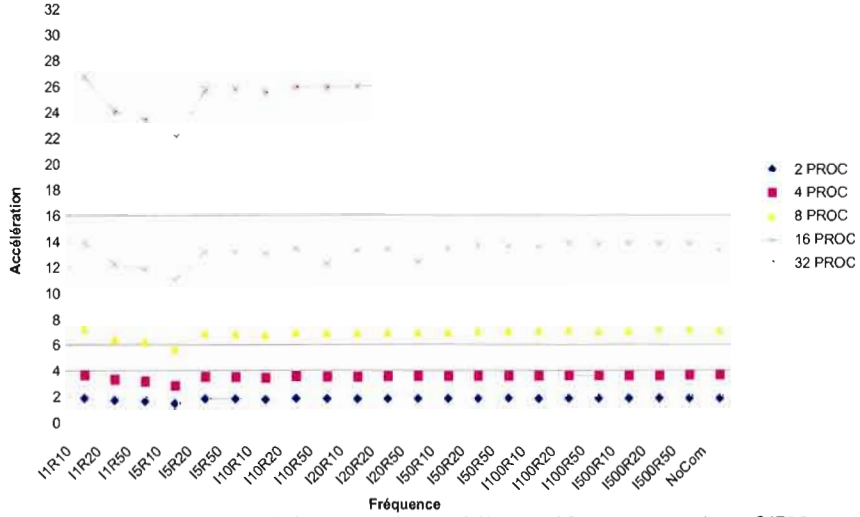

Figure 3.6 – Accélérations du problème 500 avec stratégie GEN

3.5 Conclusion

La parallélisation sans échanges d'informations a apporté des résultats peu convaincants mais a permis de poser les premières hypothèses quant à la comparaison de la performance de la stratégie de subdivision de la population par rapport à la stratégie de réduction du nombre de générations. En effet, la stratégie de réduction du nombre de générations a permis d'obtenir de meilleurs résultats et des accélérations linéaires dans la plupart des cas.

Lors de l'ajout des communications entre les processeurs sous forme d'anneau unidirectionnel, la stratégie de subdivision de population s'est avérée la plus efficace et a permis de confirmer les conclusions obtenues par d'autres auteurs [28]. D'un autre coté, les deux stratégies présentent des accélérations acceptables et aucun ralentissement par rapport à la version séquentielle n'est observé.

Pour pousser encore plus les limites de l'algorithme développé, des problèmes de grande taille ont été générés et testés. Il a été constaté que le comportement de l'algorithme pour ces problèmes n'est pas le même et que la qualité de solution et les bonnes accélérations étaient plus difficiles à obtenir. Il est également intéressant de constater que les meilleures solutions ont été générées pour des fréquences plus élevées que celles des petits problèmes. On peut ainsi déduire une relation directe entre la taille de la population et la fréquence d'échange.

La parallélisation des algorithmes génétiques est une voie intéressante pour obtenir des résultats de qualité dans des temps raisonnables. Cependant, il faut savoir définir les paramètres appropriés pour en tirer le meilleur profit. De plus, il a été démontré que l'augmentation du nombre de processeurs est bénéfique jusqu'à un certain point au-delà duquel aucune amélioration de la qualité n'est possible.

Chapitre 4

Conclusion

Ces dernières années, le domaine du parallélisme a connu un développement très important tant sur le plan matériel que logiciel. Un des exemples les plus frappant est la commercialisation des ordinateurs avec processeurs duo-coeur pour l'usage domestique et pour les entreprises. La popularisation des technologies parallèles vient combler un besoin de performance et de vitesse toujours croissant.

De manière similaire, le domaine des métaheuristiques a très nettement atteint une maturité lui permettant de se placer comme une méthode de choix pour la résolution de problèmes combinatoires. Plus particulièrement, les algorithmes génétiques ont suscité l'intérêt de la communauté scientifique par leur capacité d'adaptation à un grand nombre de problèmes et leur haut niveau de raffinement.

Ces deux domaines, à première vue distincts, viennent se rejoindre avec la conception de métaheuristiques parallèles et contribuent énormément au domaine de l'optimisation combinatoire. De manière plus spécifique, les algorithmes génétiques parallèles sont considérés parmi les métaheuristiques parallèles les plus populaires et de nombreuses recherches sont conduites en ayant recours à cette technique.

Ce mémoire visait à mettre au point une parallélisation efficace d'un algorithme génétique et a contribué à consolider cette métaheuristique en tant qu'alternative de choix pour la résolution d'un problème d'ordonnancement. D'après les résultats obtenus dans ce mémoire, la version parallèle présente un avancement au point de vue qualité et rapidité par

rapport à la version séquentielle. De plus, l'adoption de la stratégie de décomposition par division de population confirme les conclusions de la littérature.

Par rapport au premier objectif consistant à concevoir un algorithme génétique séquentiel (AG0) pour la résolution de ce problème et d'avoir des résultats comparables à l'algorithme de Rubin & Ragatz [134], on a constaté dans un premier temps des résultats de qualité inférieure de la part de l'AG0. Une modification plus poussée des paramètres de cet algorithme a permis par la suite de mettre en place une version plus performante (AG1) et de la tester sur un éventail plus large de problèmes tests. On peut alors considérer le premier objectif comme atteint puisque les résultats obtenus avec la nouvelle version sont de meilleure qualité.

Le deuxième objectif était de concevoir une version parallèle de l'algorithme génétique (AGP0) en utilisant un modèle et une topologie précise. Le plan d'expérience suivi a permis de cerner le comportement de l'algorithme lorsque celui-ci changeait de paramétrage. Même si certaines configurations ne donnaient pas de résultats satisfaisants, leur étude a permis de dégager les causes de ralentissement et de manque de diversité pour raffiner les paramètres.

Il n'a pas été possible de tester les problèmes de petite taille sur plus que 4 processeurs en raison de la conservation du nombre d'individus de la population égal au nombre de commandes. Toutefois, ceci ouvre une perspective quant à revoir la taille de population idéale pour les cas extrêmes. En effet, s'il n'a pas été possible de tester des petites populations sur un grand nombre de processeurs, il n'en demeure pas moins que l'exécution de grands problèmes sur 32 processeurs révèle qu'il y aurait sûrement une taille de population à ne pas dépasser. C'est dans cette mesure que le deuxième objectif a été atteint en identifiant les facteurs qui font que l'AGP0 performe tant sur le plan qualité que le plan de l'accélération.

L'atteinte des objectifs de ce mémoire confirme que la parallélisation des métaheuristiques en général et des algorithmes génétiques en particulier est une voie très prometteuse. L'algorithme parallèle implémenté durant ce travail prouve qu'une amélioration de temps et de qualité est possible s'il est correctement configuré.

Bibliographie

[1] Abramson, D., Mills, G., and Perkins, S. Parallelisation of a Genetic Algorithm for the Computation of Efficient Train Schedules.

[2] Adamidis, P., and Petridis, V. Co-operating Populations with Different Evolution Behaviours. *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on* (1996), 188–191.

[3] Agerwala, T., Martin, J., Mirza, J., Sadler, D., Dias, D., and Snir, M. SP2 System Architecture. *IBM Systems Journal 38*, 2/3 (1999), 414–446.

[4] Ahn, C., Goldberg, D., and Ramakrishna, R. Multiple-deme parallel estimation of distribution algorithms : Basic framework and application. *Lecture Notes in Computer Science 3019 : Parallel Processing and Applied Mathematics, PPAM 2003* (2003), 544–551.

[5] Aiex, R., Binato, S., and Resende, M. Parallel GRASP with path-relinking for job shop scheduling. *Parallel Computing 29*, 4 (2003), 393–430.

[6] Aiex, R., Resende, M., Pardalos, P., and Toraldo, G. GRASP with path relinking for three-index assignment. *INFORMS Journal on Computing 17*, 2 (2005), 224–247.

[7] Alba, E., Almeida, F., Blesa, M., Cabeza, J., Cotta, C., Diaz, M., Dorta, I., Gabarro, J., Leon, C., and Luna, J. MALLBA : A library of skeletons for combinatorial optimisation. *Proceedings of the International Euro-Par Conference, Paderborn, Germany, LNCS 2400* (2002), 927–932.

[8] Alba, E., Luna, F., and Nebro, A. Parallel Heterogeneous Genetic Algorithms for Continuous Optimization. *Parallel Computing 30*, 5-6 (2004), 699–719.

[9] Alba, E., and Luque, G. Measuring the performance of parallel metaheuristics. In *Parallel Metaheuristics : A new Class of Algorithms*. Wiley-Interscience, 2005.

[10] Alba, E., and Nebro, A. New technologies in parallelism. In *Parallel Metaheuristics A New Class of Algorithms*. Wiley-Interscience, 2005.

[11] Allahverdi, A., Gupta, J., and Aldowaisan, T. A review of scheduling research involving setup considerations. *Omega 27*, 2 (1999), 219–239.

[12] Almasi, G., and Gottlieb, A. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[13] AMDAHL, G. Validity of the single processor approach to achieving large scale computing. *AFIPS Conference Proceedings 30* (1990), 483–485.

[14] ANDERSON, E., AND FERRIS, M. *A Genetic Algorithm for the Assembly Line Balancing Problem.* University of Waterloo Press Waterloo, Ont., Canada, Canada, 1990.

[15] ARENAS, M., COLLET, P., EIBEN, A., JELASITY, M., MERELO, J., PAECHTER, B., PREUSS, M., AND SCHOENAUER, M. A framework for distributed evolutionary algorithms. *Parallel Problem Solving from Nature–PPSN VII, Proc. Seventh Int'l Conf., Granada 2439* (2002), 665–675.

[16] AYDIN, M., AND YIGIT, V. Parallel simulated annealing. In *Parallel Metaheuristics : A new Class of Algorithms*, E. Alba, Ed. Wiley-Interscience, 2005, pp. 267–287.

[17] AZENCOTT, R. *Simulated Annealing : Parallelization Techniques.* Wiley-Interscience, 1992.

[18] BACK, T., FOGEL, D., AND MICHALEWICZ, Z. *Handbook of Evolutionary Computation.* Oxford University Press, 1997.

[19] BARR, R., AND HICKMAN, B. *Reporting Computational Experiments with Parallel Algorithms : Issues, Measures, and Experts' Opinions.* Dept. of Computer Science and Engineering, Southern Methodist University, 1992.

[20] BELDING, T. The distributed genetic algorithm revisited. *Proceedings of the Sixth International Conference on Genetic Algorithms* (1995), 114–121.

[21] BENTHIN, C., WALD, I., SCHERBAUM, M., AND FRIEDRICH, H. Ray Tracing on the CELL Processor. *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), 15–23.

[22] BETHKE, A. *Comparison of Genetic Algorithms and Gradient-based Optimizers on Parallel Processors : Efficiency of Use of Processing Capacity.* The University of Michigan, College of Literature, Science, and the Arts, Computer and Communication Sciences Dept, 1976.

[23] BLAZEWICZ, J., TRYSTRAM, D., ECKER, K., AND PLATEAU, B. *Handbook on Parallel and Distributed Processing.* Springer New York, 2000.

[24] BLUM, C., ROLI, A., AND ALBA, E. An introduction to metaheuristic techniques. In *Parallel Metaheuristics : A new Class of Algorithms.* Wiley-Interscience, 2005.

[25] BUSETTI, F. Simulated annealing overview, 2004.

[26] CAHON, S., MELAB, N., AND TALBI, E. ParadisEO : A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics 10*, 3 (2004), 357–380.

[27] CANTU-PAZ, E. A survey of parallel genetic algorithms. *Calculateurs Paralleles 10*, 2 (1998), 141–171.

[28] CANTU-PAZ, E. Topologies, migration rates, and multi-population parallel genetic algorithms. *Proceedings of the Genetic and Evolutionary Computation Conference 1* (1999), 91–98.

[29] CANTU-PAZ, E. Effective and Accurate Parallel Genetic Algorithms, 2000.

[30] CANTU-PAZ, E., AND GOLDBERG, D. Efficient parallel genetic algorithms : theory and practice. *Computer Methods in Applied Mechanics and Engineering 186*, 2 (2000), 221–238.

[31] CHANDY, K., AND KESSELMAN, C. CC++ : A declarative concurrent object oriented programming notation. *Research Directions in Concurrent Object-Oriented Programming* (1993), 281–313.

[32] CHEN, D., LEE, C., PARK, C., AND MENDES, P. Parallelizing simulated annealing algorithms based on high-performance computer. *Journal of Global Optimization 39*, 2 (2007), 261–289.

[33] CRAINIC, T., TOULOUSE, M., ET AL. *Parallel Metaheuristics.* Centre for Research on Transportation. Centre de recherche sur les transports (CRT), 1998.

[34] CRAINIC, T., TOULOUSE, M., AND GENDREAU, M. Toward a Taxonomy of Parallel Tabu Search Heuristics. *INFORMS Journal on Computing 9*, 1 (1997), 61–72.

[35] CULLER, D., GUPTA, A., AND SINGH, J. *Parallel Computer Architecture : A Hardware/Software Approach.* Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1997.

[36] CUNG, V., MARTINS, S., RIBEIRO, C., AND ROUCAIROL, C. Strategies for the parallel implementation of metaheuristics. *Essays and Surveys in Metaheuristics* (2002), 263–308.

[37] DAVIDOR, Y. A Naturally Occurring Niche & Species Phenomenon : The Model and First Results. *Proc 4th International Conf on Genetic Algorithms, Morgan-Kaufmann* (1991), 257–263.

[38] DAVIS, L. Applying adaptive algorithms to epistatic domains. *Proceedings of the International Joint Conference on Artificial Intelligence 1* (1985), 161–163.

[39] DAVIS, L., ET AL. *Handbook of genetic algorithms.* Van Nostrand Reinhold New York, 1991.

[40] DE FARIA ALVIM, A., AND RIBEIRO, C. Balanceamento de Carga na Paralelização da Meta-heurística GRASP.

[41] DEBUDAJ-GRABYSZ, A., AND RABENSEIFNER, R. Nesting OpenMP in MPI to Implement a Hybrid Communication Method of Parallel Simulated Annealing on a Cluster of SMP Nodes. *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer-Verlag Berlin Heidelberg, LNCS 3666* (2005), 18–27.

[42] DELISLE, P., KRAJECKI, M., GRAVEL, M., AND GAGNE, C. Parellel implementation of an ant colony optimization metaheuristic with openmp. *International Conference of Parallel Architectures and Complication Techniques. Proceedings of the third European workshop on OpenMP* (2001), 8–12.

[43] DENNIS, J., AND MISUNAS, D. A preliminary architecture for a basic data-flow processor. *ACM SIGARCH Computer Architecture News 3*, 4 (1975), 126–132.

[44] DORIGO, M. Optimization, Learning and Natural Algorithms. *Unpublished doctoral dissertation, Dipartimento di Elettronica, Politecnico di Milano, Italy* (1992).

[45] DORIGO, M. *Ant Colony Optimization.* MIT Press, 2004.

[46] DORIGO, M., AND BLUM, C. Ant colony optimization theory : A survey. *Theoretical Computer Science 344*, 2 (2005), 243–278.

[47] DORIGO, M., AND GAMBARDELLA, L. Ant colonies for the traveling salesman problem. *BioSystems 43*, 2 (1997), 73–81.

[48] DORIGO, M., AND GAMBARDELLA, L. Ant colony system : a cooperative learning approach to the travelingsalesman problem. *Evolutionary Computation, IEEE Transactions on 1*, 1 (1997), 53–66.

[49] DUNCAN, R. A survey of parallel computer architectures. *Computer 13* (1990), 5–16.

[50] EARICKSON, J., SMITH, R., AND GOLDBERG, D. SGA-Cube : A Simple Genetic Algorithm for nCUBE 2 Hypercube Parallel Computers. *The Clearinghouse for Genetic Algorithms, Report 91005* (1991).

[51] FABER, V., LUBECK, O., AND WHITE JR, A. Superlinear speedup of an efficient sequential algorithm is not possible. *Parallel Computing 3*, 3 (1986), 259–260.

[52] FEO, T., AND RESENDE, M. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization 6*, 2 (1995), 109–133.

[53] FLYNN, M. J. Very high-speed computing systems. *Proceedings of the IEEE 54 12* (1966), 1901–1909.

[54] FOGARTY, T., AND HUANG, R. Implementing the Genetic Algorithm on Transputer Based Parallel Processing Systems. *Proceedings of the 1st Workshop on Parallel Problem Solving from Nature* (1990), 145–149.

[55] FORUM, M. P. I. Mpi : A message-passing interface standard. *International Journal of Supercomputer Applications 8(3/4)* (1994), 165–414.

[56] FOSTER, I. *Designing and building parallel programs : concepts and tools for parallel software engineering.* Addison Wesley, Reading, MA, USA, 1995.

[57] FOSTER, I. Languages for parallel processing. In *Handbook on Parallel and Distributed Processing.* Springer, 2000.

[58] FRANCA, P., MENDES, A., AND MOSCATO. P. A memetic algorithm for the total tardiness single machine scheduling problem. *European Journal of Operational Research 132*, 1 (2001), 224–242.

[59] GAGNÉ, C., GRAVEL, M., AND PRICE, W. Comparing an ACO algorithm with other heuristics for the single machine scheduling problem with sequence-dependent setup times. *Journal of the Operational Research Society 53*, 8 (2002), 895–906.

[60] GAGNÉ, C., GRAVEL, M., AND PRICE, W. Using metaheuristic compromise programming for the solution of multiple-objective scheduling problems. *Journal of the Operational Research Society 56* (2005), 687–698.

[61] GARCÍA-LÓPEZ, F., MELIÁN-BATISTA, B., MORENO-PÉREZ, J., AND MORENO-VEGA, J. The Parallel Variable Neighborhood Search for the p-Median Problem. *Journal of Heuristics 8*, 3 (2002), 375–388.

[62] GAREY, M., AND JOHNSON, D. *Computers and Intractability : A Guide to the Theory of NP-Completeness.* WH Freeman & Co. New York, NY, USA, 1979.

[63] GEN, M., AND CHENG, R. *Genetic Algorithms and Engineering Design.* Wiley-Interscience, 1997.

[64] GENDREAU, M. Recent Advances in Tabu Search. *Essays and Surveys in Metaheuristics, CC Ribeiro and P. Hansen (eds)* (2001), 369–378.

[65] GLOVER, F. Future paths for integer programming and artificial intelligence. *Computers & Operations Research 13* (1986), 533–549.

[66] GLOVER, F. Tabu search Uncharted domains. *Annals of Operations Research 149*, 1 (2007), 89–98.

[67] GLOVER, F., AND LAGUNA, M. Tabu Search.

[68] GOLDBERG, D., AND LINGLE, R. Alleles, loci, and the traveling salesman problem. *Genetic algorithms and their applications, Proc. 1st Int. Conf., Pittsburgh/PA* (1985).

[69] GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learnin.* Addison-Wesley, Reading, Mass., 1989.

[70] GONG, Y., NAKAMURA, M., AND TAMAKI, S. Parallel genetic algorithms on line topology of heterogeneous computing resources. *Proceedings of the 2005 conference on Genetic and evolutionary computation* (2005), 1447–1454.

[71] GOODMAN, E. An Introduction to GALOPPS. *East Lansing : Michigan State University* (1996).

[72] GORGES-SCHLEUTER, M. ASPARAGOS an asynchronous parallel genetic optimization strategy. *Proceedings of the third international conference on Genetic algorithms table of contents* (1989), 422–427.

[73] GRAMA, A., GUPTA, A., KARYPIS, G., AND KUMAR, V. *Introduction to parallel computing.* Addison-Wesley New York, 2003.

[74] GREFENSTETTE, J. Parallel adaptive algorithms for function optimization. *Vanderbilt University, Nashville, TN, Tech. Rep. CS-81-19* (1981).

[75] GREFENSTETTE, J., GOPAL, R., ROSMAITA, B., AND VAN GUCHT, D. Genetic Algorithms for the Traveling Salesman Problem. *Proceedings of the 1st International Conference on Genetic Algorithms table of contents* (1985), 160–168.

[76] GROSSO, P. *Computer Simulations of Genetic Adaptation : Parallel Subcomponent Interaction in a Multilocus Model.* PhD thesis, University of Michigan, 1985.

[77] GUO, Q. Parallel genetic algorithms for the solution of inverse heat conduction problems. *International Journal of Computer Mathematics 84*, 2 (2007), 241–249.

[78] GUPTA, S., AND SMITH, J. Algorithms for single machine total tardiness scheduling with sequence dependent setups. *European Journal of Operational Research 175*, 2 (2006), 722–739.

[79] HANSEN, P., AND MLADENOVIC, N. Variable neighborhood search for the P-median. *Location Science 5*, 4 (1997), 207–226.

[80] HAUSER, R., AND MANNER, R. Implementation of standard genetic algorithm on MIMD machines. *Parallel Problem Solving fron Nature, PPSN 3* (1994), 504–513.

[81] He, H., Sỳkora, O., Salagean, A., and Mäkinen, E. Parallelisation of genetic algorithms for the 2-page crossing number problem. *Journal of Parallel and Distributed Computing 67*, 2 (2007), 229–241.

[82] Herrera, F., and Lozano, M. Gradual distributed real-coded genetic algorithms. *Evolutionary Computation, IEEE Transactions on 4*, 1 (2000), 43–63.

[83] Herrera, F., Lozano, M., and Verdegay, J. Tackling Real-Coded Genetic Algorithms : Operators and Tools for Behavioural Analysis. *Artificial Intelligence Review 12*, 4 (1998), 265–319.

[84] Holland, J. Adaptation in Neural and Artificial Systems. *Ann Arbor, MI : University of Michigan Press* (1975).

[85] Homberger, J., and Gehring, H. A two-phase hybrid metaheuristic for the vehicle routing problem with time windows. *European Journal of Operational Research 162*, 1 (2005), 220–238.

[86] InfiniBand, SM. Trade Association. *InfiniBand TM Architecture Specification Volume 1.*

[87] JáJá, J. *An introduction to parallel algorithms.* Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 1992.

[88] Karp, A., and Flatt, H. Measuring parallel processor performance. *Communications of the ACM 33*, 5 (1990), 539–543.

[89] Kirkpatrick, S., Gelatt Jr, C., and Vecchi, M. Optimization by Simulated Annealing. *Science 220*, 4598 (1983), 671.

[90] Koulamas, C. The Total Tardiness Problem : Review and Extensions. *Operations Research 42*, 6 (1994), 1025–1041.

[91] Laguna, M. *Scatter Search : Methodology and Implementation in C.* Kluwer Academic Publishers, 2003.

[92] Laudon, J., and Lenoski, D. The SGI Origin : A ccnuma Highly Scalable Server. *Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on* (1997), 241–251.

[93] Lee, Y., Bhaskaran, K., and Pinedo, M. A heuristic to minimize the total weighted tardiness with sequence-dependent setups. *IIE Transactions 29*, 1 (1997), 45–52.

[94] Levine, D. Users Guide to the PGAPack Parallel Genetic Algorithm Library. *Argonne National Laboratory 95*, 18 (1995), 1–77.

[95] Lin, S., Punch III, W., and Goodman, E. Coarse-grain parallel genetic algorithms : categorization and newapproach. *Parallel and Distributed Processing. 1994. Proceedings. Sixth IEEE Symposium on* (1994), 28–37.

[96] LUO, X., and CHU, C. A branch-and-bound algorithm of the single machine schedule with sequence-dependent setup times for minimizing maximum tardiness. *European Journal of Operational Research 180*, 1 (2007), 68–81.

[97] Luo, X., Chu, C., and Wang, C. Some dominance properties for single-machine tardiness problems with sequence-dependent setup. *International Journal of Production Research 44*, 17 (2006), 3367–3378.

[98] LUQUE, G., ALBA, E., AND DORRONSORO, B. Parallel genetic algorithms. In *Parallel Metaheuristics : A new Class of Algorithms*. Wiley-Interscience, 2005.

[99] MACK, D., BORTFELDT, A., AND GEHRING, H. A parallel hybrid local search algorithm for the container loading problem. *International Transactions in Operational Research 11*, 5 (2004), 511–533.

[100] MALONY, A. *Tools for parallel Computing : A Performance Evaluation Perspective*. Springer, 2000, ch. VII, p. 342.

[101] MANFRIN, M., BIRATTARI, M., STUTZLE, T., AND DORIGO, M. Parallel ant colony optimization for the traveling salesman problem. *Ant Colony Optimization and Swarm Intelligence, 5th International Workshop, ANTS 4150* (2006).

[102] MARTINS, S., RESENDE, M., RIBEIRO, C., AND PARDALOS, P. A Parallel Grasp for the Steiner Tree Problem in Graphs Using a Hybrid Local Search Strategy. *Journal of Global Optimization 17*, 1 (2000), 267–283.

[103] MEJIA-OLVERA, M., AND CANTU-PAZ, E. DGENESIS-software for the execution of distributed genetic algorithms. *Proceedings of the XXLatinoamericanConferenceonComputerScience. Monterrey, Mexico* (1994), 935–946.

[104] MENDES, R., PEREIRA, J., AND NEVES, J. A Parallel Architecture for Solving Constraint Satisfaction Problems. *Proceedings of Metaheuristics Int. Conf 2* (2001), 109–114.

[105] MENDIBURU, A., LOZANO, J., AND MIGUEL-ALONSO, J. Parallel Estimation of Distribution Algorithms : New approaches. Tech. rep., Technical Report EHU-KAT-IK-1-3, Department of Computer Architecture and Technology, The University of the Basque Country, 2003.

[106] METROPOLIS, N., ROSENBLUTH, A., ROSENBLUTH, M., TELLER, A., AND TELLER, E. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics 21* (2004), 1087.

[107] MICHALEWICZ, Z. *Genetic Algorithms+ Data Structures= Evolution Programs*. Springer, 1996.

[108] MICHEL, R., AND MIDDENDORF, M. An Island Model Based Ant System with Lookahead for the Shortest Supersequence Problem. *Proceedings of the 5th International Conference on Parallel Problem Solving from Nature* (1998), 692–701.

[109] MIDDENDORF, M., REISCHLE, F., AND SCHMECK, H. Multi Colony Ant Algorithms. *Journal of Heuristics 8*, 3 (2002), 305–320.

[110] MORENO-PÉREZ, J., HANSEN, P., AND MLADENOVIC, N. Parallel variable neighborhood search. In *Parallel Metaheuristics : A new Class of Algorithms*. Wiley-Interscience, 2005.

[111] MÜHLENBEIN, H., MAHNIG, T., AND RODRIGUEZ, A. Schemata, Distributions and Graphical Models in Evolutionary Optimization. *Journal of Heuristics 5*, 2 (1999), 215–247.

[112] MUHLENBEIN, H., AND PAASS, G. From recombination of genes to the estimation of distributions I. Binary parameters. *Lecture Notes in Computer Science 1141* (1996), 178–187.

[113] MÜHLENBEIN, H., SCHOMISCH, M., AND BORN, J. The parallel genetic alghorithm as function optimizer. *Parallel computing 17*, 6-7 (1991), 619–632.

[114] NOWLABS OHIO STATE UNIVERSITY. Mvapich2 toolset. (http ://nowlab.cse.ohiostate.edu/projects/mpi-iba/).

[115] OLIVER, I., SMITH, D., AND HOLLAND, J. A study of permutation crossover operators on the traveling salesman problem. *in : Proc. 2nd Int. Conf on Genetic Algorithms Massachusetts Institute of Technology, Cambridge, MA* (1987), 224–230.

[116] OPENMP, A. OpenMP FORTRAN API. Tech. rep., Version 1.1. Technical report, http ://www. openmp. org, November 1999, 1999.

[117] PARDALOS, P., PITSOULIS, L., AND RESENDE, M. A parallel GRASP implementation for the quadratic assignment problem. *Parallel Algorithms for Irregularly Structured Problems–Irregular 94* (1994), 115–133.

[118] PARKINSON, D. Parallel efficiency can be greater than unity. *Parallel Computing 3*, 3 (1986), 261–262.

[119] PETTEY, C., LEUZE, M., AND GREFENSTETTE, J. A parallel genetic algorithm. *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application table of contents* (1987), 155–161.

[120] PITSOULIS, L., AND RESENDE, M. Greedy randomized adaptive search procedures. *Handbook of Applied Optimization* (2002), 168–183.

[121] PLATEAU, B., AND TRYSTRAM, D. Parallel and distributed computing : State-of-the-art ans emerging trends. In *Handbook on Parallel and Distributed Processing*. Springer, 2000.

[122] POTTS, J., GIDDENS, T., AND YADAV, S. The development and evaluation of an improved genetic algorithmbased on migration and artificial selection. *Systems, Man and Cybernetics, IEEE Transactions on 24*, 1 (1994), 73–86.

[123] POTVIN, J. Genetic algorithms for the traveling salesman problem. *Annals of Operations Research 63*, 3 (1996), 337–370.

[124] QF MANUGEMENT, D. Global optimization for artificial neural networks : A tabu search application. *European Journal of Operational Research 106* (1998), 570–584.

[125] QUINN, M. J. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, 2003.

[126] RADCLIFFE, N., AND SURRY, P. The reproductive plan language RPL2 : Motivation, architecture and applications. *Genetic Algorithms in Optimisation, Simulation and Modelling* (1999), 65–94.

[127] RAGATZ, G. Scheduling to minimize tardiness on a single machine with sequence dependent setup times. *Opns Res 23* (1989), 118–136.

[128] RAGATZ, G. A branch-and-bound method for minimum tardiness sequencing on a single processor with sequence dependent setup times. *Proceedings : Twenty-fourth Annual Meeting of the Decision Sciences Institute* (1993), 1375–1377.

[129] RAHOUAL, M., HADJI, R., AND BACHELET, V. Parallel ant system for the set covering problem. *Proceedings of the Third International Workshop on Ant Algorithms, ANTS* (2002), 262–267.

[130] RANDALL, M., AND LEWIS, A. A Parallel Implementation of Ant Colony Optimization. *Journal of Parallel and Distributed Computing 62*, 9 (2002), 1421–1432.

[131] RIBEIRO, C., AND ROSSETI, I. Efficient parallel cooperative implementations of GRASP heuristics. *Parallel Computing 33*, 1 (2007), 21–35.

[132] RIBEIRO FILHO, J., TRELEAVEN, P., AND ALIPPI, C. *Genetic-algorithm programming environments*, vol. 27. 1994.

[133] ROBBINS, G. EnGENEer–The Evolution of Solutions. *Proceedings of the 5th Annual Seminar on Neural Networks and Genetic Algorithms* (1992).

[134] RUBIN, P. A., AND RAGATZ, G. L. Scheduling in a sequence dependent setup environment with genetic search. *Computers and Operations Research Vol. 22, No. 1* (1994), 85–99.

[135] SCHERR, A. An Analysis Of Time-Shared Computer Systems.

[136] SCHWARTZ, J. *Ultracomputers*, vol. 2. 1980.

[137] SNYDER, L. A Taxonomy of Synchronous Parallel Machines. *Proceedings of the 1988 International Conference on Parallel Processing* (1988), 281–285.

[138] STENDER, J. *Parallel Genetic Algorithms : Theory and Applications*. IOS Press, 1993.

[139] STUTZLE, T. *Parallelization Strategies for Ant Colony Optimization*, vol. 24. Kluwer Academic Publishers, 1998, pp. 87–100.

[140] STUTZLE, T., AND HOOS, H. MAX-MIN Ant System and local search for the traveling salesmanproblem. *Evolutionary Computation, 1997., IEEE International Conference on* (1997), 309–314.

[141] SUNDERAM, V., ET AL. PVM : A Framework for Parallel Distributed Computing. *Concurrency Practice and Experience 2*, 4 (1990), 315–339.

[142] SYSWERDA, G. Schedule optimization using genetic algorithms. *Handbook of Genetic Algorithms* (1991), 332–349.

[143] TAILLARD, É., GAMBARDELLA, L., GENDREAU, M., AND POTVIN, J. Adaptive memory programming : A unified view of metaheuristics. *European Journal of Operational Research 135*, 1 (2001), 1–16.

[144] TALBI, E., GEIB, J., HAFIDI, Z., AND KEBBAL, D. MARS : An adaptive parallel programming environment. *High Performance Cluster Computing 1* (1999), 722–739.

[145] TALBI, E., ROUX, O., FONLUPT, C., AND ROBILLARD, D. Parallel ant colonies for combinatorial optimization problems. *Parallel and Distributed Processing 11* (1999), 239–247.

[146] TAN, K., AND NARASIMHAN, R. Minimizing Tardiness on a Single Processor with Sequence-dependent Setup Times : a Simulated Annealing Approach. *Omega 25*, 6 (1997), 619–634.

[147] TAN, K.-C., NARASIMHAN, R., RUBIN, P., AND RAGATZ, G. A comparison of four methods for minimizing total tardiness on a single processor with sequence dependent setup times. *Omega Volume 28, Number 5* (2000), 609–609.

[148] TANESE, R. Distributed genetic algorithms. *Proceedings of the third international conference on Genetic algorithms table of contents* (1989), 434–439.

[149] TOMASEVIC, M., AND MILUTINOVIC, V. Hardware approaches to cache coherence in shared-memory multiprocessors, Part. *Micro, IEEE 14*, 5 (1994).

[150] VON EICKEN, T., CULLER, D., GOLDSTEIN, S., AND SCHAUSER, K. *Active messages : a mechanism for integrated communication and computation.* ACM Press New York, NY, USA, 1992.

[151] VON NEUMANN, J. The computer and the brain. *ACM Classic Books Series* (1958), 98.

[152] WANG, Z., WONG, Y., AND RAHMAN, M. Development of a parallel optimization method based on genetic simulated annealing algorithm. *Parallel Computing 31*, 8 (2005), 839–857.

[153] WHITLEY, D., AND STARKWEATHER, T. GENITOR II. : a distributed genetic algorithm. *Journal of Experimental & Theoretical Artificial Intelligence 2*, 3 (1990), 189–214.

[154] WHITLEY, D., STARKWEATHER, T., AND FUQUAY, D. 'A.(1989). Scheduling problems and traveling salesman : the genetic edge recombination operator. *Proceedings of the 3rd International Conference on Genetic Algorithms ICGA* (1989), 133–140.

[155] YAMAMURA, M., ONO, T., AND KOBAYASHI, S. Character-preserving genetic algorithms for traveling salesman problem. *Journal of Japanese Society for Artificial Intelligence 7*, 6 (1992), 1049–1059.

[156] YEH, J., WU, T., AND CHANG, J. Parallel genetic algorithms for product configuration management on PC cluster systems. *The International Journal of Advanced Manufacturing Technology 31*, 11 (2007), 1233–1242.